Technical Report 70-107
NGR 21-002-206
NGL 21-002-008

January 1970

A Methodology for Unified Hardware-Software Design

Yaohan Chu
Oliver R. Pardo
Jeffrey Yeh

UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER
COLLEGE PARK, MARYLAND

Technical Report 70-107
NGR 21-002-206
NGL 21-002-008

January 1970

A Methodology for Unified Hardware-Software Design

Yaohan Chu
Oliver R. Pardo
Jeffrey Yeh

This research was supported in part by Grants NGR 21-002-206 and NGL 21-002-008 from the National Aeronautics and Space Administration to the Computer Science Center of the University of Maryland.

Table of Contents

Abstract

This report describes a methodology for unifying the hardware and software design of a digital computer by mean of the Computer Design Language (or CDL). The methodology is presented in three examples: (a) finding the largest number among n given numbers, (b) buffer allocation in an input-output control system, and (c) translation of relocatable code to executable code. The algorithms in these examples are all obtained from computer programs. Important steps of the methodology are shown in great details. These are: description of the configuration by the CDL declaration statements, translation of the flow chart into the sequence chart, representation of timing and control signals, implementation by microprogram control, description of sequential operations by the CDL execution statements, production of microprogram, and simulation of the design on the CDL Simulator.

2. A Methodology

By means of the CDL, a methodology has been developed for unifying the hardware and software design. This methodology is shown by the diagram in Figure 1. As shown, a piece of software or computer program is studied and its algorithm is extracted and presented preferrably in the form of flow chart. Then, configuration for hardware implementation of the algorithm is conceived, and the flow chart is converted into the sequence chart. Both the configuration and the sequence charts are next described by the CDL statements. This description is punched into a deck of cards and simulated by the CDL Simulator (16) for checking out the implementation. The result of the siumlation is then used for evaluating the design. If the design is not satisfactory, the algorithm is modified or even replaced, and another design cycle follows. When the result becomes satisfactory, the design is then documented. Documentation in the CDL is relatively simple.

This report describes three examples whose algorithms were extracted from the existing computer programs. Each example illustrates various steps of the methodology. The first example is implemented by the sequential logic control, while the second and the third examples by microprogram control.

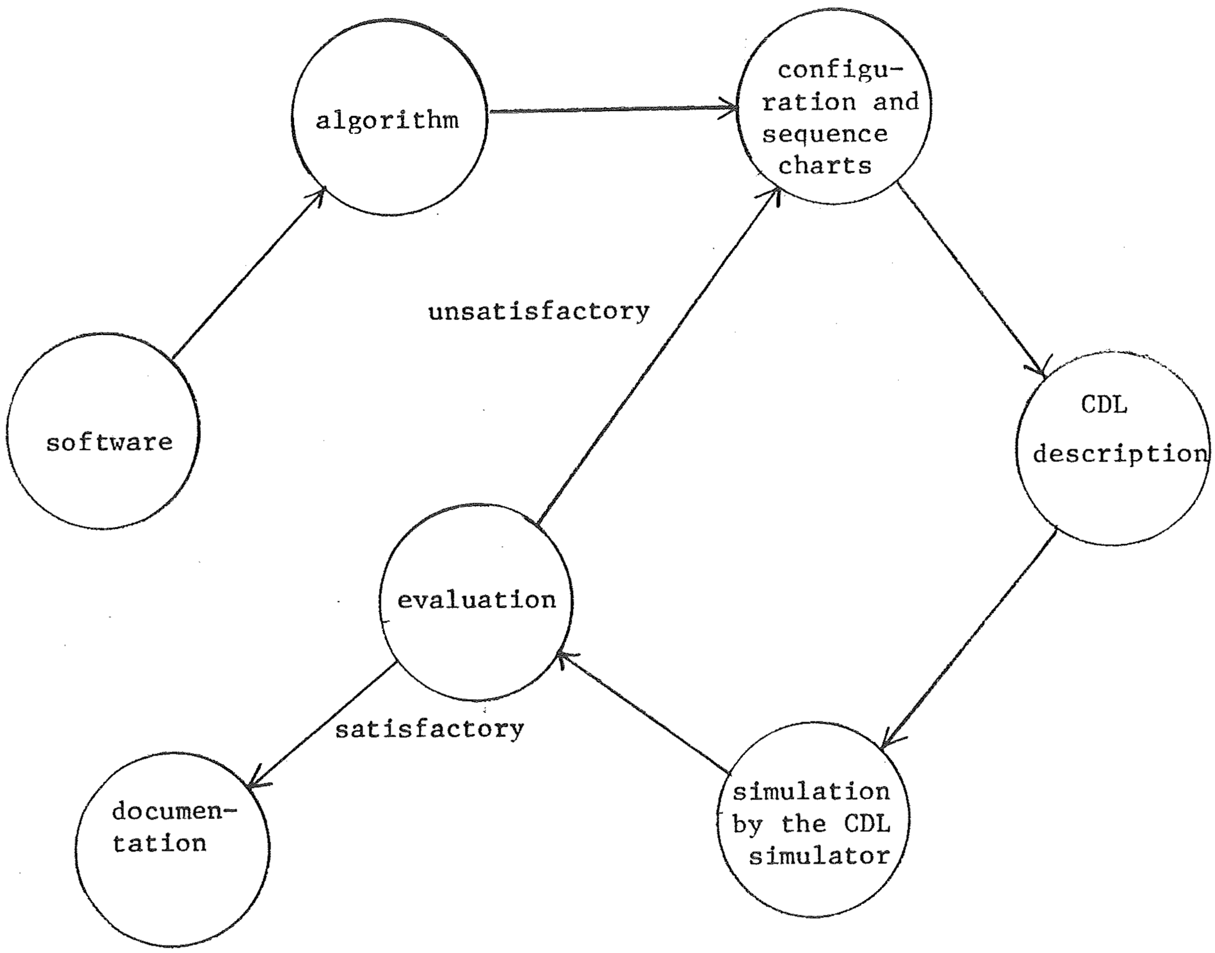

Fig. 1 Block diagram illustrating the methodology of unified hardware-software design

3. Finding the largest number

The first example is to find the largest number among n unsigned binary number. This simple example is selected for the purpose of introducing the CDL and the simulation by the CDL simulator.

3.1 Algorithm

An algorithm to find the largest number among given binary numbers X(1),...,X(n) by programming is shown in Fig.2 where n is the number of elements, m is the current largest element, k is the pointer whick points to the element now in comparison, and j is the pointer which points to the current largest element. As shown in Fig. 2, the first comparison is between elements X(n) and X(n-1) from which the larger is stored in m. The next comparison is between m and X(n-2), between m and X(n-3) and so forth where m always stores the larger element after each comparison. Pointer k begins from (n-1) and is decremented after each comparison. The finding process terminates when k reaches 0.

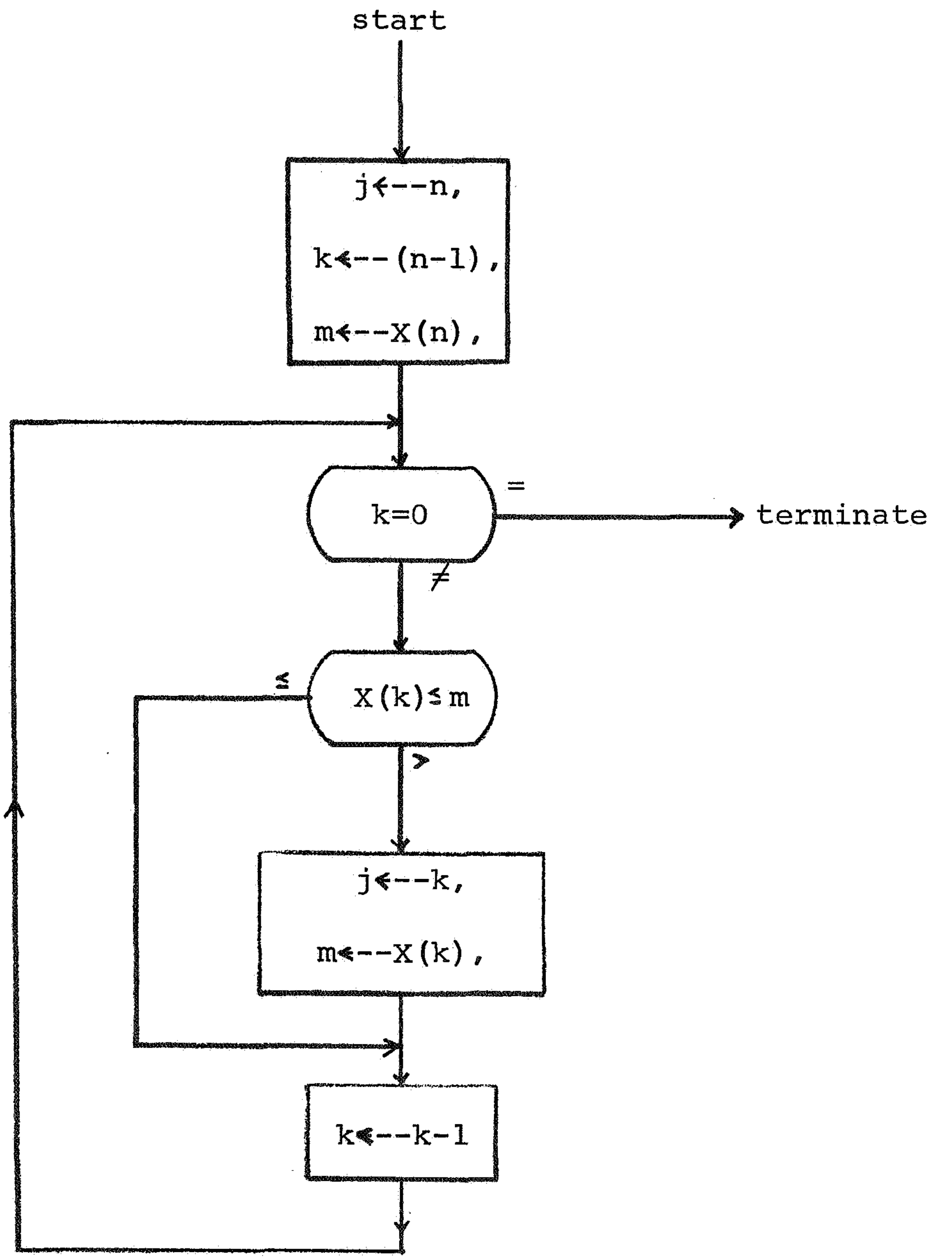

Figure 2 Flow chart of finding the largest number among n numbers

3.2 Configuration and Sequence Chart

Let the given unsigned binary integers be stored in memory X with address register C and buffer register R. Let the capacity of memory X be 1024 words and the word length be 24 bits. Assume that number n is stored in the first location and the integers in the succeeding locations of the memory. Register J and K store respectively pointers J and K. Register A stores the current largest integer after each comparison. Comparison is done by a paralled subtracter. In addition, there are control register T, switch START and light FINI. These elements are shown in the block diagram of Fig. 3.

The process of finding the largest element is shown in the sequence chart of Fig. 4. After initialization during which register C is reset to 0 and light FINI is turned to the OFF condition, the first word is read out of memory X; this word contains number n. Number n is then transferred to memory address register C so as to read out the last elememt of the given n elements; this element is stored in register A. The last second element is next read out of memory X and stored in the buffer register R. The numbers in registers A and R are compared by the parallel subtracter. Terminal BOR(0) is 1, this indicates that the unsigned binary integer in register A is smaller than that in register R. In this case, the larger number in register R is transferred to register A and the memory address where this larger number is stored in memory X is transferred to register J. The next element is then taken out of the memory and compared with the number in register R. Again, the larger number and its memory address are stored in register A and J respectively. This

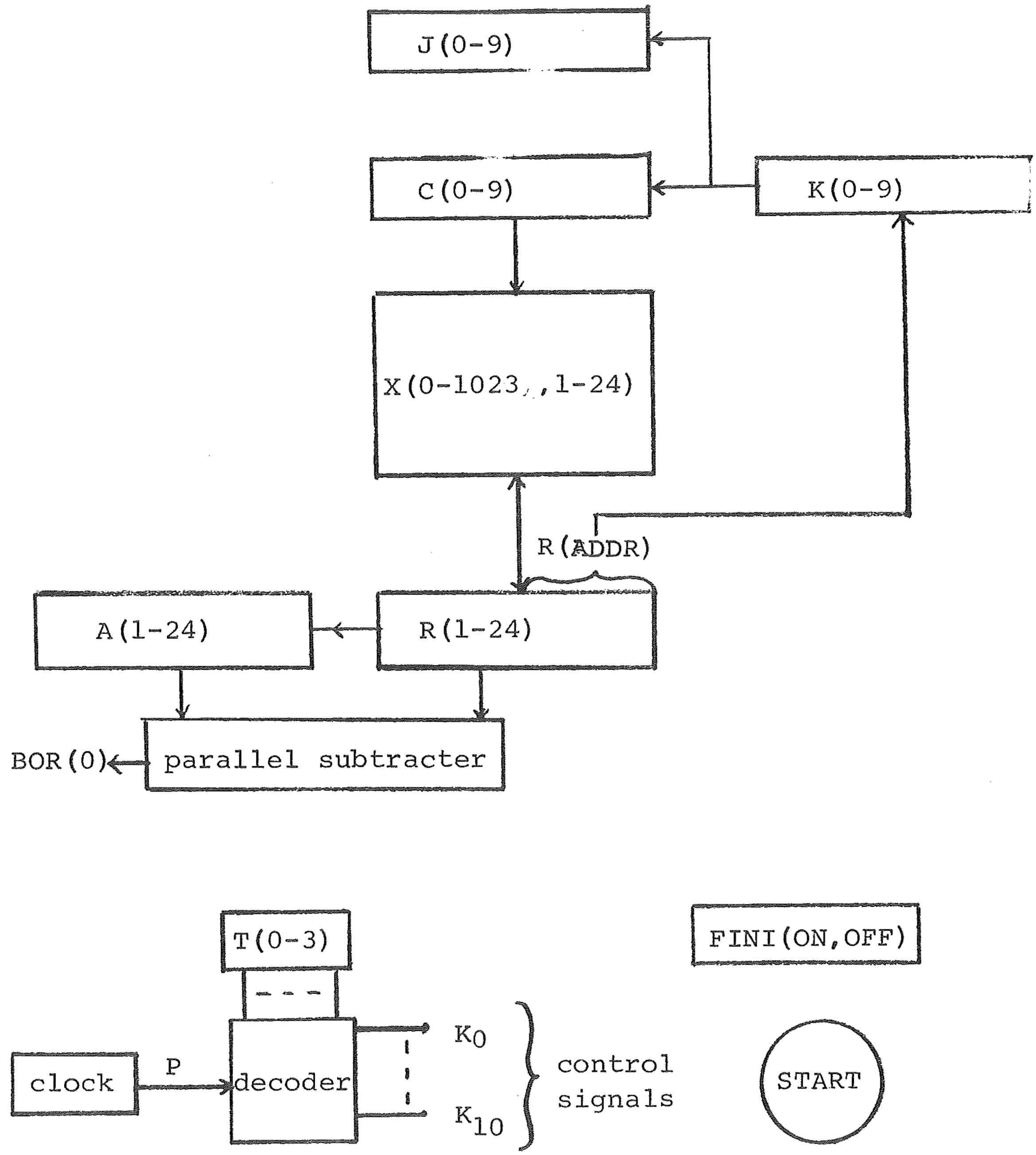

Figure 3 Configuration for finding the largest number

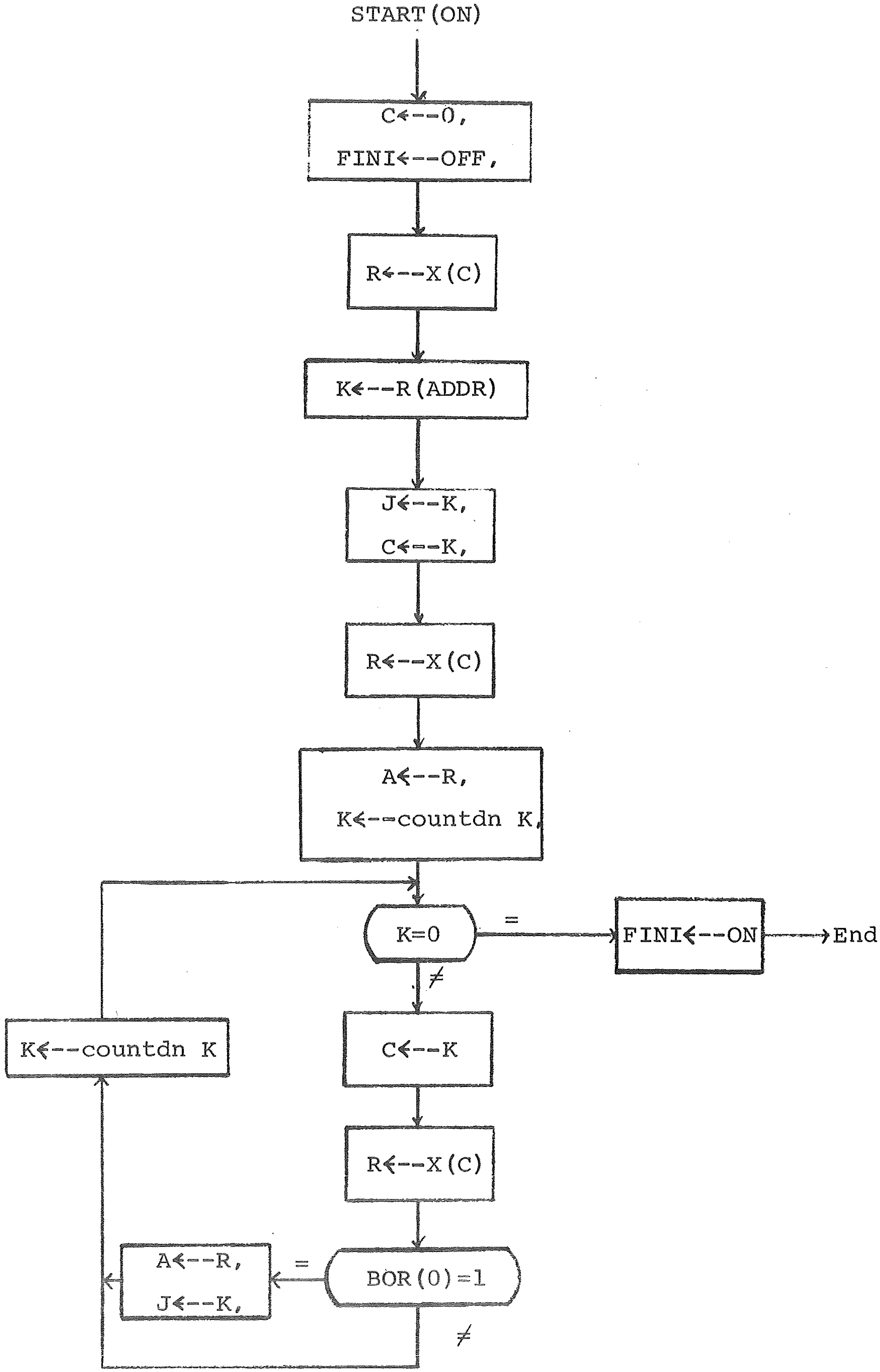

Figure 4 Sequence chart of finding the largest number

process continues until K reaches 0; at that time, all the elements are compared. The largest element is in register A and its memory address is register J.

3.3 Statement Description

The above configuration and sequence chart for finding the largest element is now described by the CDL statements.

```
Comment, configuration                                          (1)
Register,      C(0-9),        $address register
               R(1-24),       $buffer register
               J(0-9),        $store pointer j
               K(0-9).        $store pointer k
               A(1-24).       $store current largest element
               T(0-3),        $control register
Subregister,   R(ADDR)=R(15-24),
Memory,        X(C)=X(0-1023, 1-24),
Decoder,       KT(0-10)=T
Switch,        START(ON),
Light,         FINI(ON, OFF),
Terminal,      DIFF(1-24)=A(1-24)⊕R(1-24)⊕BOR(1-24),
               BOR(0-23)=A(1-24)*R(1-24)'+R(1-24)'*BOR(1-24)+BOR(1-24)*A(1-24),
               BOR(24)=0,
Clock,         P,

Comment, here begins the comparison sequence.
/START(ON)/    C<-0, FINI<-OFF, T<-0,
/KT(0)*P/      R<-X(C), T<-1,         $read out n
/KT(1)*P/      K<-R(ADDR), T<-2,      $store n in K
/KT(2)*P/      C<-K, J<-R, T<-3,      $store n in J and C
```

```
/KT(3)*P/     R<-X(C), T<-4,        $read out X(n)
/KT(4)*P/     A<-R,                 $store X(n)in A
              K<-countdn K,         $obtain (n-1)
              T<-5,
/KT(5)*P/     IF (K=0) THEN (T<-10) ELSE (T<-6)
/KT(6)*P/     C<-K, T<-7,
/KT(7)*P/     R<-X(c), T<-8         $read next element X(C)
/KT(8)*P/     IF (BOR(0)=1) THEN (A<-R, J<-K), T<-9
/KT (9)*P/    K<-contdn K, T<-5,
/KT(10)*P/    FINI<-ON,
              END
```

In the above description, the ternimal statement describes the parallel subtractor. Terminals DIFF are the different outputs of the subtracter. They are not needed, but are included for the sake of completeness.

3.4 Simulation

Statement description 1, when punched into a deck of cards, is shown in the listing of Fig. 5. The first 12 lines and the last line represent the system control cards. The last second through eighth lines represent the simulation control cards. The listing in Fig. 5 and statement description 1 are essentially identical except the slight differences in the statements and operator "countdn" being defined as a special operator.

The first part of the simulation result is shown in Fig. 6. The contents of registers C, J, K, T, A and R as well as those at memory locations 0 through 8 are tabulated after the START switch is turned to the ON position as well as at the end of the first through seventh clock cycles.

```
$IBSYS
$*      MOUNT TAPE 1090 ON A9, RING OUT AND SAVE
$*      SAVE                              THANK YOU
$PAUSE
$ATTACH          A9
$AS              SYSLB4
$REWIND          SYSLB4
$EXECUTE         USER
$ID    CHU*001/01/187*1M*50P$
$CDL3
$TRANSLATE
*MAIN
COMMENT, CONFIGURATION
 REGISTER,       C(0-11),
1                R(1-30),
1                J(0-11),
1                K(0-11),
1                A(1-30),
1                T(0-3)
 SUBREGISTER,    R(ADDR)=R(17-30)
 MEMORY,         X(C)=X(0-77,1-30)
 DECODER,        KT(0-12)=T
 SWITCH,         START(ON),
1                FINI(ON,OFF)
 CLOCK,          P
 TERMINAL,       BOR30=0,
1       BOR27=BOR30*(A(30)'+R(30))+R(30)*A(30)',
1       BOR26=BOR27*(A(27)'+R(27))+R(27)*A(27)',
1       BOR25=BOR26*(A(26)'+R(26))+R(26)*A(26)',
1       BOR24=BOR25*(A(25)'+R(25))+R(25)*A(25)'
 TERMINAL,
1       BOR23=BOR24*(A(24)'+R(24))+R(24)*A(24)',
1       BOR22=BOR23*(A(23)'+R(23))+R(23)*A(23)',
1       BOR21=BOR22*(A(22)'+R(22))+R(22)*A(22)',
1       BOR20=BOR21*(A(21)'+R(21))+R(21)*A(21)'
 TERMINAL,
1       BOR17=BOR20*(A(20)'+R(20))+R(20)*A(20)',
1       BOR16=BOR17*(A(17)'+R(17))+R(17)*A(17)',
1       BOR15=BOR16*(A(16)'+R(16))+R(16)*A(16)',
1       BOR14=BOR15*(A(15)'+R(15))+R(15)*A(15)'
 TERMINAL,
1       BOR13=BOR14*(A(14)'+R(14))+R(14)*A(14)',
1       BOR12=BOR13*(A(13)'+R(13))+R(13)*A(13)',
1       BOR11=BOR12*(A(12)'+R(12))+R(12)*A(12)',
1       BOR10=BOR11*(A(11)'+R(11))+R(11)*A(11)'
 TERMINAL,
1       BOR07=BOR10*(A(10)'+R(10))+R(10)*A(10)',
1       BOR06=BOR07*(A(07)'+R(07))+R(07)*A(07)',
1       BOR05=BOR06*(A(06)'+R(06))+R(06)*A(06)',
1       BOR04=BOR05*(A(05)'+R(05))+R(05)*A(05)'
 TERMINAL,
1       BOR03=BOR04*(A(04)'+R(04))+R(04)*A(04)',
1       BOR02=BOR03*(A(03)'+R(03))+R(03)*A(03)',
1       BOR01=BOR02*(A(02)'+R(02))+R(02)*A(02)'
 TERMINAL,
1       BOR00=BOR01*(A(01)'+R(01))+R(01)*A(01)'
```

Fig. 5 Listing of the CDL deck for simulating the example of finding the largest element

```
COMMENT, HERE BEGINS THE COMPARISON SEQUENCE.
 /START(ON)/   C=0, FINI=OFF, T=0
 /KT(0)*P/     P=X(C), T=1
 /KT(1)*P/     K=R(ADDR), T=2
 /KT(2)*P/     C=K, J=K, T=3
 /KT(3)*P/     P=X(C), T=4
 /KT(4)*P/     A=P,
               K=K.CONTDN.   ,
               T=5
 /KT(5)*P/     IF(K.EQ.0) THEN (T=12) ELSE (T=6)
 /KT(6)*P/     C=K, T=7
 /KT(7)*P/     P=X(C), T=10
 /KT(10)*P/    IF( POROO.EQ.1) THEN (A=P, J=K), T=11
 /KT(11)*P/    K=K.CONTDN., T=5
 /KT(12)*P/    FINI=ON
               END
*OPERATOR,     Y(0-11).CONTDN.
 //            Y(0-11).SUP.1    , RETURN
 END
$SIMULATE
*OUTPUT      CLOCK(1)=C,R,J,K,A,T,X(0),X(1),X(2),X(3),X(4),X(5),X(6),
                         X(7),X(10)
*SWITCH      1,START=ON
*LOAD
 X(0-10)=10,5,21,15,20,25,56,35,40
 A=0,J=0,K=0,P=0
*SIM         100,10
$RESTORE
```

Fig. 5, continued

OUTPUT OF SIMULATION

```
SWITCH INTERRUPT
    START  = ON
       C = ..0000         J = ..0000         K = ..0000          T = ....00
       A = ....00000000          R = ....00000000    000000 = ....00000010    000001 = ....00000005    000002 = ....00000021
  000003 = ....00000015   000004 = ....00000020    000005 = ....00000025    000006 = ....00000056    000007 = ....00000035
  000010 = ....00000040
*************************************************************
LABEL CYCLE    1                TRUE LABELS                CLOCK TIME =    0
                              /KT(0)*P/
       C = ..0000         J = ..0000         K = ..0000          T = ....01
       A = ....00000000          R = ....00000010    000000 = ....00000010    000001 = ....00000005    000002 = ....00000021
  000003 = ....00000015   000004 = ....00000020    000005 = ....00000025    000006 = ....00000056    000007 = ....00000035
  000010 = ....00000040
*************************************************************
LABEL CYCLE    2                TRUE LABELS                CLOCK TIME =    1
                              /KT(1)*P/
       C = ..0000         J = ..0000         K = ..0010          T = ....02
       A = ....00000000          R = ....00000010    000000 = ....00000010    000001 = ....00000005    000002 = ....00000021
  000003 = ....00000015   000004 = ....00000020    000005 = ....00000025    000006 = ....00000056    000007 = ....00000035
  000010 = ....00000040
*************************************************************
LABEL CYCLE    3                TRUE LABELS                CLOCK TIME =    2
                              /KT(2)*P/
       C = ..0010         J = ..0010         K = ..0010          T = ....03
       A = ....00000000          R = ....00000010    000000 = ....00000010    000001 = ....00000005    000002 = ....00000021
  000003 = ....00000015   000004 = ....00000020    000005 = ....00000025    000006 = ....00000056    000007 = ....00000035
  000010 = ....00000040
*************************************************************
LABEL CYCLE    4                TRUE LABELS                CLOCK TIME =    3
                              /KT(3)*P/
       C = ..0010         J = ..0010         K = ..0010          T = ....04
       A = ....00000000          R = ....00000040    000000 = ....00000010    000001 = ....00000005    000002 = ....00000021
  000003 = ....00000015   000004 = ....00000020    000005 = ....00000025    000006 = ....00000056    000007 = ....00000035
  000010 = ....00000040
*************************************************************
LABEL CYCLE    5                TRUE LABELS                CLOCK TIME =    4
                              /KT(4)*P/
       C = ..0010         J = ..0010         K = ..0007          T = ....05
       A = ....00000040          R = ....00000040    000000 = ....00000010    000001 = ....00000005    000002 = ....00000021
  000003 = ....00000015   000004 = ....00000020    000005 = ....00000025    000006 = ....00000056    000007 = ....00000035
  000010 = ....00000040
*************************************************************
LABEL CYCLE    6                TRUE LABELS                CLOCK TIME =    5
                              /KT(5)*P/
       C = ..0010         J = ..0010         K = ..0007          T = ....06
       A = ....00000040          R = ....00000040    000000 = ....00000010    000001 = ....00000005    000002 = ....00000021
  000003 = ....00000015   000004 = ....00000020    000005 = ....00000025    000006 = ....00000056    000007 = ....00000035
  000010 = ....00000040
*************************************************************
LABEL CYCLE    7                TRUE LABELS                CLOCK TIME =    6
                              /KT(6)*P/
       C = ..0007         J = ..0010         K = ..0007          T = ....07
       A = ....00000040          R = ....00000040    000000 = ....00000010    000001 = ....00000005    000002 = ....00000021
  000003 = ....00000015   000004 = ....00000020    000005 = ....00000025    000006 = ....00000056    000007 = ....00000035
  000010 = ....00000040
*************************************************************
```

Fig. 6 The first part of the simulation result of the example of finding the largest element

4. Buffer Allocation

The second example is buffer allocation which is taken from the GETBUF routine of the Simple Input Output Control System (SIOCS) now being developed. This example illustrates the hardware implementation of the input output control system of a microprogrammed operating system.

4.1 Problem description

In the SIOCS, double buffers are used for each file except when the user employs his own buffer scheme. When a file is being opened or redefined, the SIOCS searches the available buffer chains, finds two buffers of proper size from one of the buffer chains and assigns them to that file. When these two buffers are no longer used, the SIOCS releases them and returns them to the available buffer chains.

The GETBUF routine of the SIOCS performs the function of obtaining a double buffers from the available buffer chain and then assigning them to the file. An example of the available buffer chain is shown in Fig. 7. The entry of this chain is in the Available-buffer-chain Entry Table or ABC Entry Table. The LINK field of this entry as shown in Fig. 7 contains the address of the first buffer of this buffer chain, while the SIZE field of this entry describes the size of the buffer. All buffers are initially linked in the available buffer chains, and the ABC Entry Table contains all the entries for the available buffer chains, each for the buffers of one size.

The GETBUF routine is called by the GETBUF macro instruction whose format is shown in Fig. 8, where FILENAME is the name of the file to which the buffers are assigned. To use this routine, the file must be previously opened; this means that FILENAME must be the symbolic address of a File Control Block (FCB). The format of a FCB, as an example, is shown in Fig. 9, where the parameter

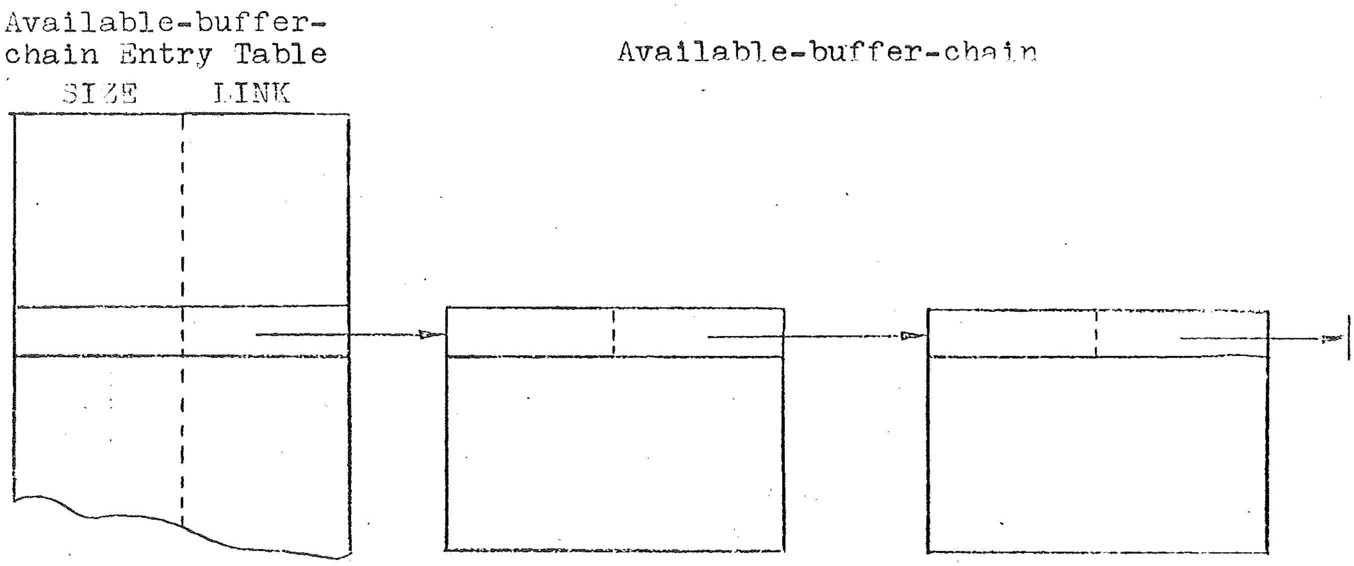

Fig.7 Structure of an Available-buffer-chain

Fig.8 Format of the GETBUF macro instruction

names are explained in Table 1. Note that the Unit Control Block mentioned in Table 1 is a table which stores the status of an input or an output device. Pointer IOBUF is the pointer which always points to the current input (or output) buffer where the data are being read into (or being sent out), while pointer PROBUF is the pointer which always points to the processing buffer, from which the data currently being processed are obtained. All the other fields, such as OC, CRTCL, BUSY, EOF, TYPE, AVBCT and UCB address are not used by this GETBUF routine, and are thus not further described.

The inputs to the GETBUF routine are: the address of the FCB, the buffer size in the FCB, the address of the ABC Entry Table, the ABC Entry Table and the available buffer chains. The outputs from the routine are the two buffer addresses which are placed in the third word of the FCB and in accumulator A. If no buffers are available, accumulator A is returned with its contents being 0.

4.2 Algorithm

The algorithm for buffer allocation is shown in the flow chart of Fig. 10, where the symbolic names are defined or explained in Table 2. As shown, the algorithm beings by obtaining the buffer size from the fourth word of the File Control Block (see Fig. 9) or

BUFSZE<--SIZE(FILENAME+3)

It then proceeds to searching the ABC Entry Table, where P is a pointer which scans the entries of the table. When the search finds the entry of that available buffer chain with the buffer size equal to the desired buffer size (i.e. SIZE(P)=BUFSZE), the two addresses of the first two buffers (pointed by the pointer Q) of the chain are obtained and stored in the left and right half of accumulator A. The buffers are removed from the chain and the chain is

<table>
<tr><td colspan="6">FILE NAME</td></tr>
<tr><td>OC</td><td>TYPE</td><td></td><td colspan="3">UCB ADDRESS</td></tr>
<tr><td colspan="3">IOBUF</td><td colspan="3">PROBUF</td></tr>
<tr><td colspan="2">SIZE</td><td>CRTCL</td><td>BUSY</td><td>EOF</td><td>AVBCT</td></tr>
</table>

Fig.9 Format of the File Control Block
(see text for explaination)

Table 1, Terms of the FCB in Fig. 9

Term	Explanation
TYPE	type of the file
OC	an open/close indicator
UCB address	address of the Unit Control Block (UCB) of a device
IOBUF	a pointer for the input (or output) buffer
PROBUF	a pointer for the processing buffer
SIZE	buffer size
CRTCL	critical number of the processing buffer
BUSY	buffer busy indicator
EOF	end-of-file indicator
AVBCT	a counter which counts the available words remaining in the processing buffer

Table 2, Symbolic names of the algorithm

Symbolic name	Explanation
BUFSZE	a variable which represents the buffer size of a file
P	a pointer which scans the Available-buffer-chain Entry Table
Q	a pointer which scans an available buffer chain
FILENAME	symbolic location of the FCB. (a known quantity)
TABLE	symbolic location of the Available-buffer-chain Entry Table (a known quantity)
SIZE(X)	SIZE field of the word pointed by pointer X
LINK(X)	LINK field of the word pointed by pointer X
WORD(X)	a word pointed by pointer X
A	accumulator
A(0-17)	left half of the accumulator
A(18-35)	right half of the accumulator
←--	a symbol which denotes "assign to" or "replaced by"

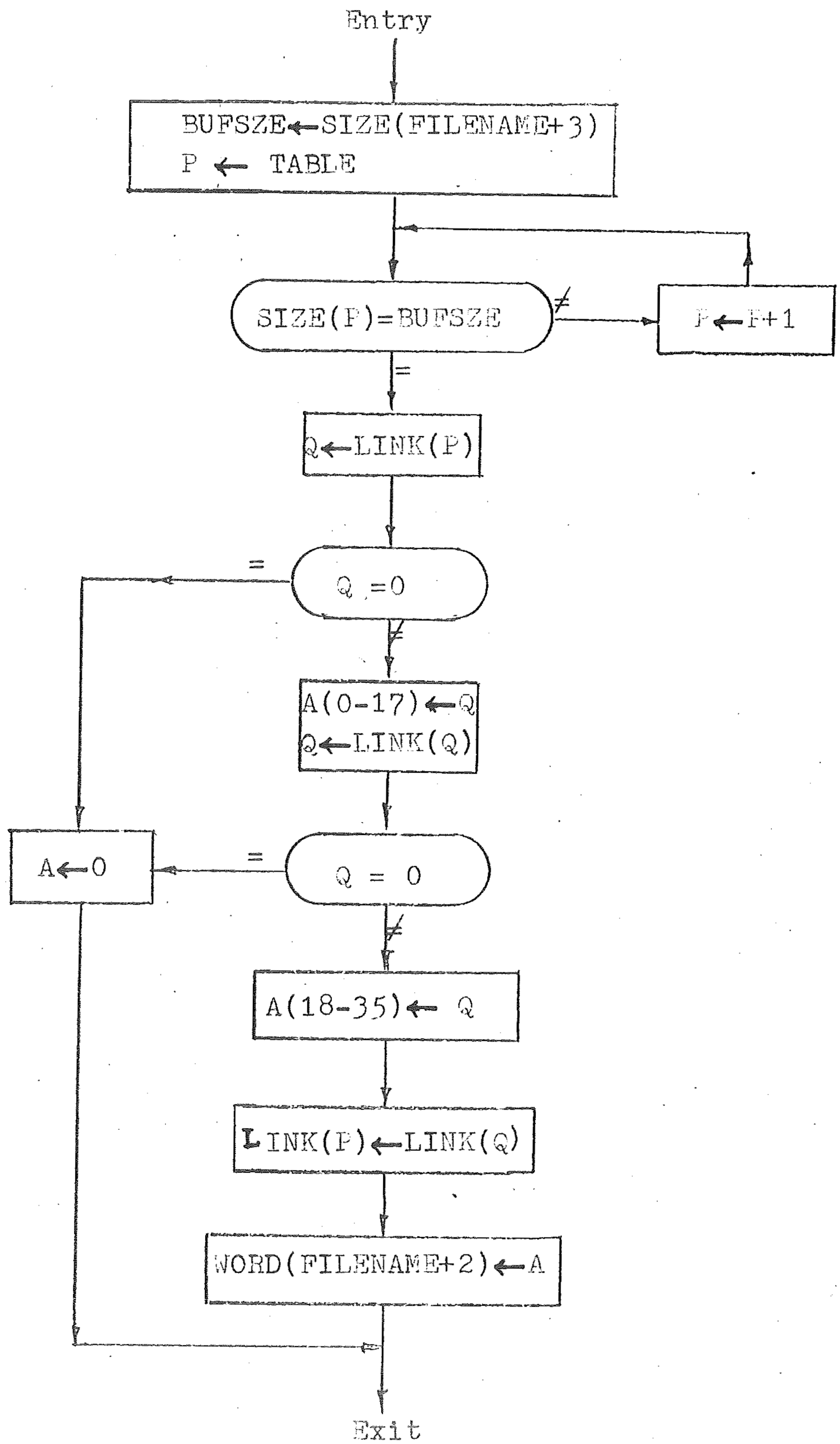

Fig.10 Flow Chart for buffer allocation

again properly linked by putting LINK(Q) into LINK(P). These two buffer addresses are next placed in the proper location of the File Control Block. If the search results in finding no two buffers of the proper size, accumulator A is reset to 0. In either case, the allocation is terminated.

As an example, consider an ABC Entry Table and two available buffer chains as those shown in Fig. 11 where the numbers are the initial values. Let FILEA be the symbolic address of the File Control Block shown in Fig. 12, where 0's in the first and third fields of word FILEA+1 indicates no information and 1 in the second field indicates that this file is a tape file. Number 20 in word FILEA+3 indicates that buffer size of the file is of 20 memory words. When the algorithm is executed, it finds that the buffer size required by the file is 20-word and that the buffers with this size are available from the table entry at location 102 in Fig. 11. The LINK fields of the buffers of the available buffer chain are then scanned beginning from the entry, and two such buffers whose addresses 201 and 271 in Fig. 11 are found. These addresses are next placed in the third word of the File Control Block (i.e. FILEA+2) in Fig. 14 and the available buffer chain is again linked after the removal of these two buffers. The output after allocation is shown in Figs. 13 and 14. Notice that the two fields of the third word of FCB in Fig. 12 are changed to 201 and 271 in Fig. 14 respectively. Meanwhile, in the ABC Entry Table of Fig. 11, the Link field 201 for the buffer chain of size 20 in Fig. 11 is changed to 291 as shown in Fig. 13.

As another example, assume that the contents of the ABC Entry Table and the available buffer chains in Fig. 13 which are the outputs from the first example are used as the initial values of the second example. Let FILEB be the symbolic address of a File Control Block shown in Fig. 15. When the GETBUF routine is being executed, the buffer size is found from the FCB in Fig. 15

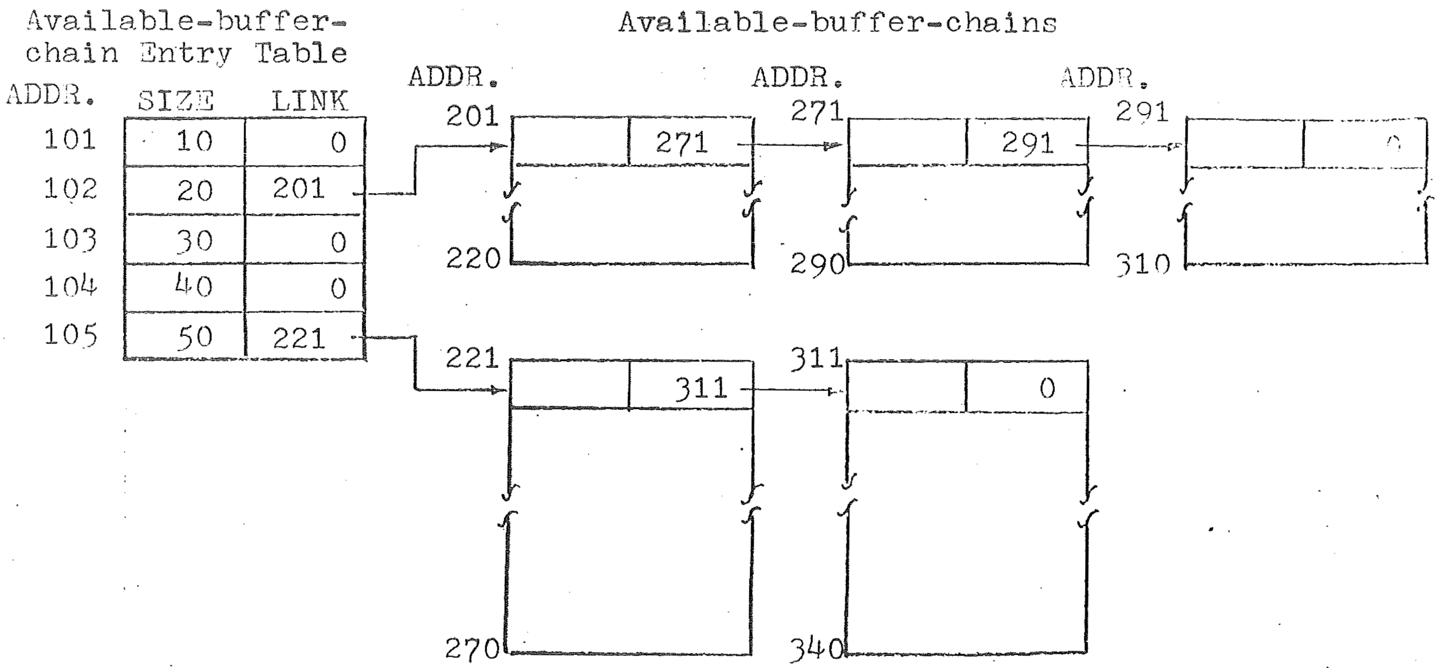

Fig.11 An example of the Available-buffer-chain Entry Table and Available-buffer-chains

File Control Block(FCB)

<table>
<tr><th>ADDR.</th><th colspan="6"></th></tr>
<tr><td>FILEA</td><td colspan="6">FILE NAME</td></tr>
<tr><td>FILEA+1</td><td>0</td><td>1</td><td>0</td><td colspan="3">UCB ADDRESS</td></tr>
<tr><td>FILEA+2</td><td colspan="3">0</td><td colspan="3">0</td></tr>
<tr><td>FILEA+3</td><td colspan="2">20</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Fig.12 FCB with address FILEA (input)

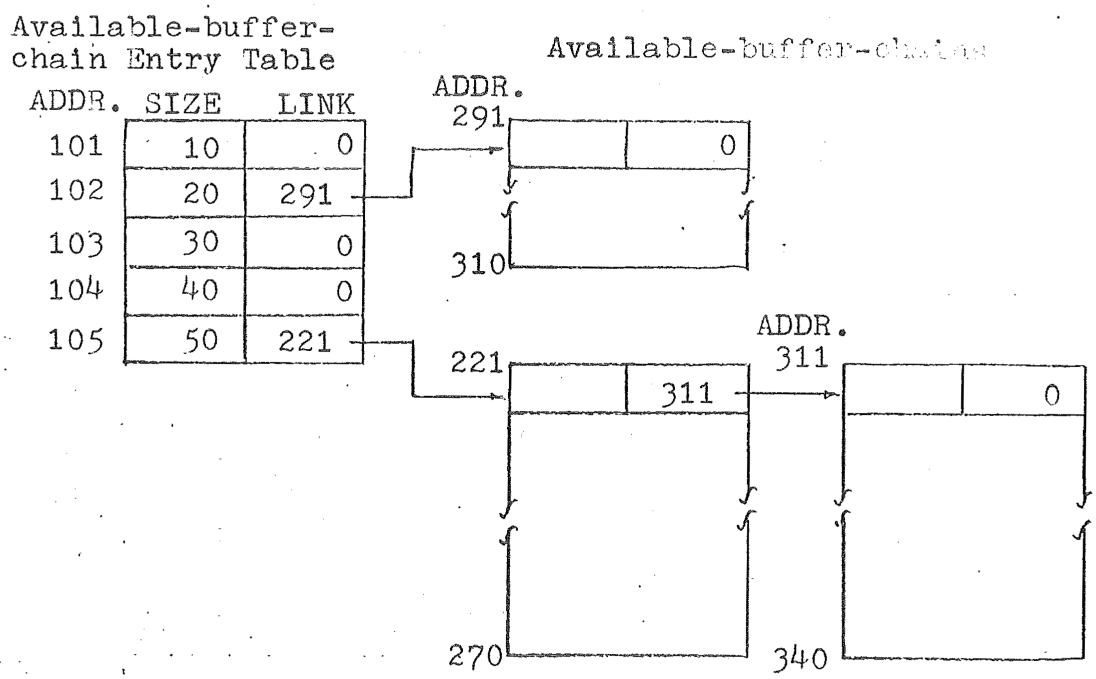

Fig.13 The Available-buffer-chain Entry Table and Available-buffer-chains (output of Example 1)

ADDR. File Control Block (FCB)

ADDR.					
FILEA	FILE NAME				
FILEA+1	0	1	0	UCB ADDRESS	
FILEA+2	201			271	
FILEA+3	20	0	0	0	0

Fig.14 FCB with address FILEA (output)

File Control Block (FCB)

<table>
<tr><td>FILEB</td><td colspan="6">FILE NAME</td></tr>
<tr><td>FILEB+1</td><td>0</td><td>1</td><td>0</td><td colspan="3">UCB ADDRESS</td></tr>
<tr><td>FILEB+2</td><td colspan="3">0</td><td colspan="3">0</td></tr>
<tr><td>FILEB+3</td><td colspan="2">40</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Fig.15 FCB with address FILEB

to be 40. The entry for the buffer chain of this size is found from the ABC Entry Table in Fig. 13. Since its LINK field is 0, this means that there is no buffer of this size available, and accumulator A is reset to 0. Therefore, the contents of the File Control Block, the ABC Entry Table and the available buffer chain remain unchanged.

4.3 Configuration

The computer elements that are required for implementing the buffer allocation algorithm are shown in the block diagram of Fig. 16 except the control part to be described subsequently. As shown, there is a random-access memory M where the File Control Block, the ABC Entry Table and the available buffer chains are located. The memory has a capacity of 32,768 36-bit words with a 15-bit address register MAR and a 36-bit register MB. There are two 15-bit registers FILENAME and ENTRY, a 9-bit register BUFSZE, a 36-bit accumulator A, and two single-bit register READ and WRITE in addition to a 9-bit parallel comparator and a 15-bit parallel adder. Register FILENAME stores the address of the FCB of the given file. Register ENTRY stores a pointer which scans the entries of the ABC Entry Table. Register BUFSZE stores the buffer size of the given file. The accumulator is where the two buffer addresses are assembled. The comparator compares for equality between the contents of register BUFSZE and those of the leftmost 9-bits of the accumulator. The adder adds the contents of register FILENAME to those of register MAR and the resulting sum is placed in register MAR. Registers READ and WRITE are used for activating respectively the read and write operations of the memory.

The above configuration for the processing unit is now described by the following CDL declaration statements.

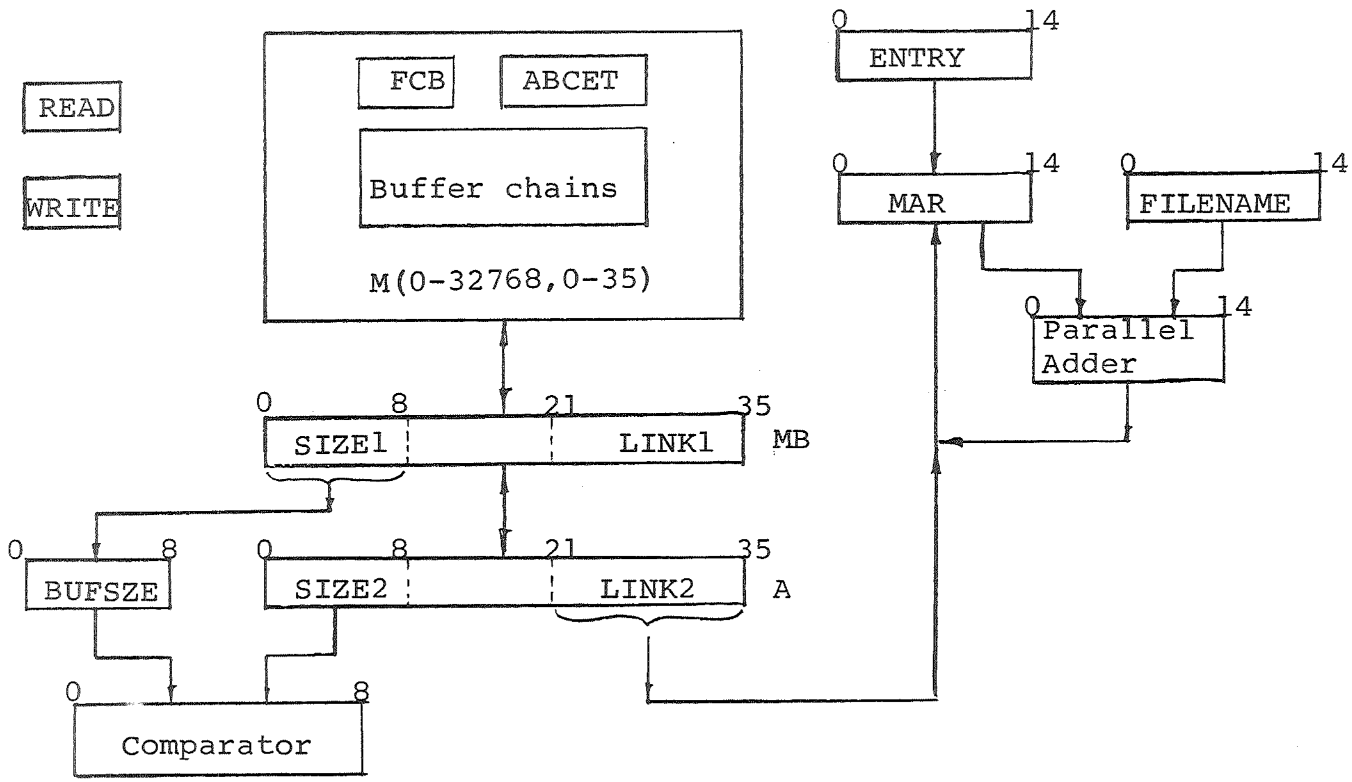

Fig. 16 Configuration of the processing unit

```
Comment, Configuration of the processing unit.                (4.2)

Register,      MAR(0-14),          $address register
               MB(0-35),           $buffer register
               ENTRY(0-14),        $pointer to scan Entry Table
               FILENAME(0-14),     $store the address of the FCB
               A(0-35),            $store two buffer addresses
               BUFSZE(0-8),        $store the desired buffer size
               READ,               $READ control register
               WRITE,              $WRITE control register

Memory,        M(MAR)=M(0-32768,0-35),

Subregister,   MB(SIZE1)=MB(0-8),  $SIZE field
               MB(LINK1)=MB(21-35),  $LINK field
               A(SIZE2)=A(0-8),    $SIZE field
               A(LINK2)=A(21-35),  $LINK field

Terminal,      UNEQUL(0-8)=A(0-8)⊕BUFSZE,  $comparator
```

4.4 Sequence charts

The sequence operations of the buffer allocation are shown in the sequence chart in Fig. 17. It is assumed (a) the formats of the File Control Block, the ABC Entry Table, and the available buffer chains are those in Figs. 9 and 7, (b) the File Control Block, the ABC Entry Table and the available buffer chains are initially stored in the main memory, (c) the address of the FCB of the given file is initially stored in register FILENAME, and (d) the address of the ABC Entry Table is stored in register ENTRY.

As shown in Fig. 17, the address FILENAME is first incremented by 3 and the sum is then transferred to register MAR. A word is next read out of the memory into buffer register MB which now stores the fourth word of the FCB.

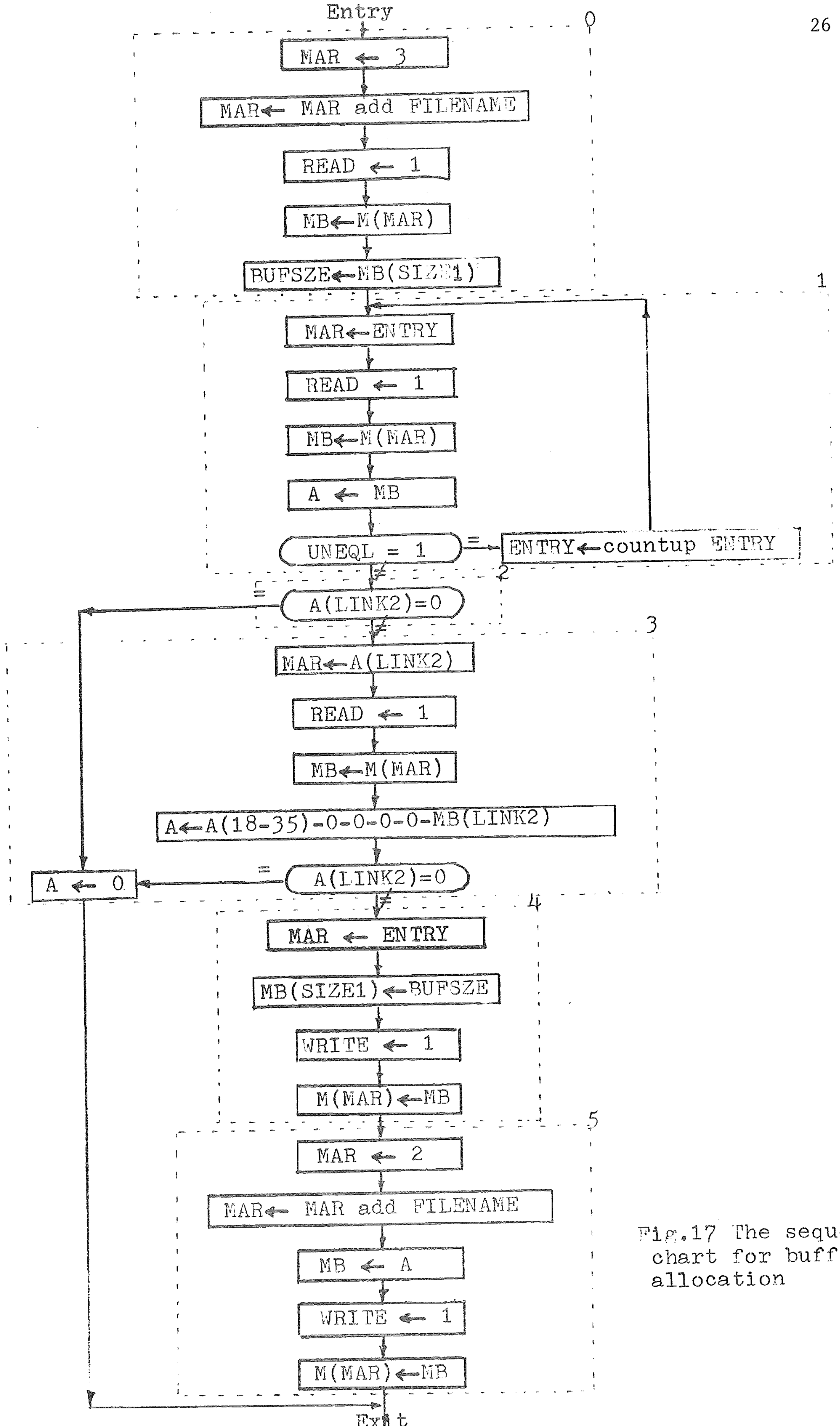

Fig.17 The sequence chart for buffer allocation

The SIZE field of this word is transferred to register BUFSZE. These micro-operations are shown in Block 0 in Fig. 17. (See block number at top-right corner of each block.) In block 1 there is a loop searching for the entries in the ABC Entry Table to find an entry whose SIZE field is equal to the contents of BUFSZE. Block 2 tests availability of the buffers in the chain. If no buffer is available as indicated by the LINK field of A being 0, accumulator A is reset to 0. Otherwise, the two addresses of the two buffers are stored in register A as shown in block 3. Finally the ABC Entry Table is updated as shown in block 4, and the two buffer addresses are stored in the third word of the FCB of the file in block 5. At this time, the sequence is terminated.

4.5 Microprogram Control Configuration

The control memory is a small but fast memory having a capacity of 255 36-bit words with an 8-bit address register CAR and a 36-bit buffer register F. Each word in the control memory is called a control word or a micro-instruction. In the control memory is stored a microprogram which consists of a series of micro-instructions. Single-bit register E indicates the fetch (when 1) or the execution (when 0) of each micro-instruction. There is a four-phase clock P(0-3). Each main memory cycle is chosen consisting of four control memory cycles and each control memory cycle coincides with one clock cycle. Register MC is used to sequence the four control memory cycles in each main memory cycle. Register RUN is used for indicating the start (when 1) and the stop (when 0) status of the machine. Switch START is selected for manual control of the start operation of the machine. The block diagram in Fig. 18 shows the configuration of the control unit.

The above configuration may also be described by the following CDL state-

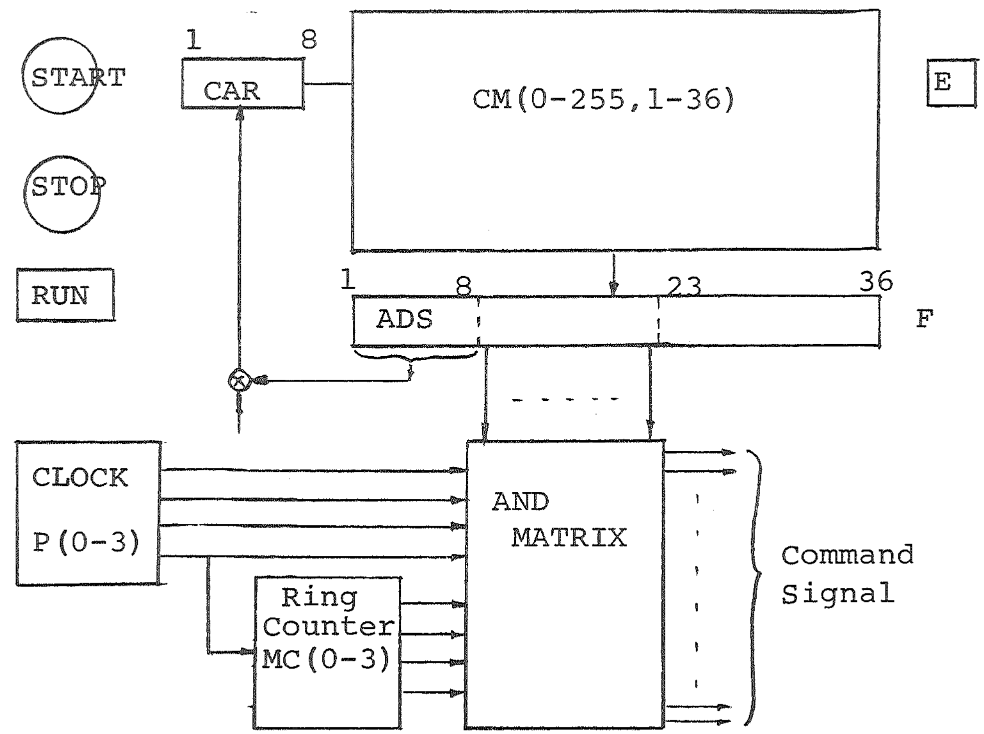

Fig.18 Configuration of the control unit

ments.

```
Comment, configuration for the microprogram control
Register,     MC(0-3),        $register for sequencing main memory cycles
              CAR(1-8),       $control memory address register
              F(1-36),        $control word register
              E,              $CM fetch-execution control register
              RUN,            $start-stop register
Subregister,  F(ADS)=F(1-8),  $address field
Memory,       CM(CAR)=CM(0-255,1-36),
Switch,       START(ON)       $start switch
Comment, each control memory cycle coincides with one clock cycle and each
         main memory cycle coincides with four control memory cycles
Clock,        P(0-3)          $four-phase clock
Comment, sequencing register MC as a ring counter.
/P(3)*RUN/    MC←cir MC
```

In the above description, each of the four bits of register MC represent each of the four control memory cycles in one main memory cycle, and the sequencing of the four control memory cycles is accomplished by making register MC to function as a ring counter.

4.6 Timing and Control Signals

Each main memory cycle is chosen to consist of four control memory cycles, and each control memory cycle coincides with each clock cycle. Therefore, there are 4 steps in each control memory cycle and 16 steps in each main memory 16 steps in each main memory cycle. The control signals for these 16 steps are described by the following sequence of 16 labels, Comment, description of the labels.

/MC(0)*P(0)*RUN/ $beginning of a main and a control memory cycle

/MC(0)*P(1)*RUN/

/MC(0)*P(2)*RUN/

/MC(0)*P(3)*RUN/ $end of a control memory cycle

/MC(1)*P(0)*RUN/ $beginning of a control memory cycle

/MC(1)*P(1)*RUN/

/MC(1)*P(2)*RUN/

/MC(1)*P(3)*RUN/ $end of a control memory cycle

/MC(2)*P(0)*RUN/ $beginning of a control memory cycle

/MC(2)*P(1)*RUN/

/MC(2)*P(2)*RUN/

/MC(2)*P(3)*RUN/ $end of a control memory cycle

/MC(3)*P(0)*RUN/ $beginning of a control memory cycle

/MC(3)*P(1)*RUN/

/MC(3)*P(2)*RUN/ E--0

/MX(3)*P(3)*RUN/ $End of both memory cycles

As shown in the above labels, the four steps in each control memory cycle are controlled by the four phases of clock P(0-3), and the four control memory cycles in each main memory cycle are controlled by the four states of ring counter MC(0-3). Register RUN controls the generation of the sequence of the 16 control signals in a main memory cycle as indicated in Fig. 18.

During each main memory cycle, the data is read out of or written into the main memory. It is now specified that the transfer of the main memory address to register MAR and the initiation of the main memory read or write must occur during the second step (i.e. /MC(0)*P(1)*RUN/). For a read operation, the word is available at buffer register MB during the sixth step (i.e. /MC(1)*P(1)*RUN/). For a write operation the word to be stored into the memory is transfered into buffer register MB before the 12th step (/MC(2)*P(3)*RUN/).

If certain micro-operations occur in control memory cycle, the following sequence of four labels is used,

```
/P(0)*RUN*E'/      F<-CM(CAR)         $beginning of a control memory cycle
                   E<-1
/P(1)*RUNE'/
/P(2)*RUN*E'/
/P(3)*RUN*E'/      CAR<-countup CAR   $end of a control memory cycle
```

In the above sequence of labels, register E is used to control the advance or stop of the 4 steps in a control memory cycle. When register E contains a 0, the sequence of the labels exist; otherwise, it disappears.

Whenever the register E is set to 0, a micro-instruction is read out of the control memory. It is now specified that the incrementing of the control memory address register CAR and the initiation of the control memory read must occur during clock phase P(3) of the preceeding control memory cycle, and the control word becomes available at buffer register F during the first clock phase P(0) of the current control memory cycle. Micro-operations activated by the micro-instruction in register F are executed between the first clock phase P(0) of the current control memory cycle and the initiation for the reading out of the next micro-instruction.

Note that, at the step with label /P(0)*RUN*E'/, register E is set to 1 at the same time a control word is transfered into register F. While the register is reset to 0 at the last step of each main memory cycle (i.e. /MC(3)*P(2)*RUN/); this causes the fetch of the next control word at the beginning of the next main memory cycle

4.7 Control word format

The format of the control word is shown in Fig. 19. It consists of three fields, the address field, the control-bit field and the constant field. The address field contains a control memory address for micro-branching. The constant field contains a constant. Each bit of the control-bit field controls one or more micro-operations which are shown in Fig. 19. The assignment of the control bits to the micro-operations is arbitrarily made.

4.8 Statement description

With the timing and control signals as well as the control word format being established, it is now possible to describe the control signals by means of labels for each micro-operation in the sequence chart of Fig. 15. Each label is a logical AND of the timing signal, the clock signal and a control bit. With the labels, execution statements can now be written for each micro-operation or a group of micro-operations. And each block in Fig. 15 becomes one micro-instruction.

Each block in the sequence chart of Fig. 15 requires one main memory cycle, and the micro-operations in the block are then assigned to the previously described timing and control signals (i.e., labels). The micro-operations in each block is translated into one micro-instruction. In this manner, the sequence chart can be described by the following CDL statements.

```
Comment, the buffer allocation sequence begins here                 (6)
Comment, start or stop operation.
/START(ON)/          RUN<--1,MC<--1,MAR<--3,
                     ENTRY<--100,CAR<--0,E<--0,
Comment, fetch the micro-instruction at location 0.
/P(0)*RUN*E'/        F<--CM(CAR),E<--1,
```

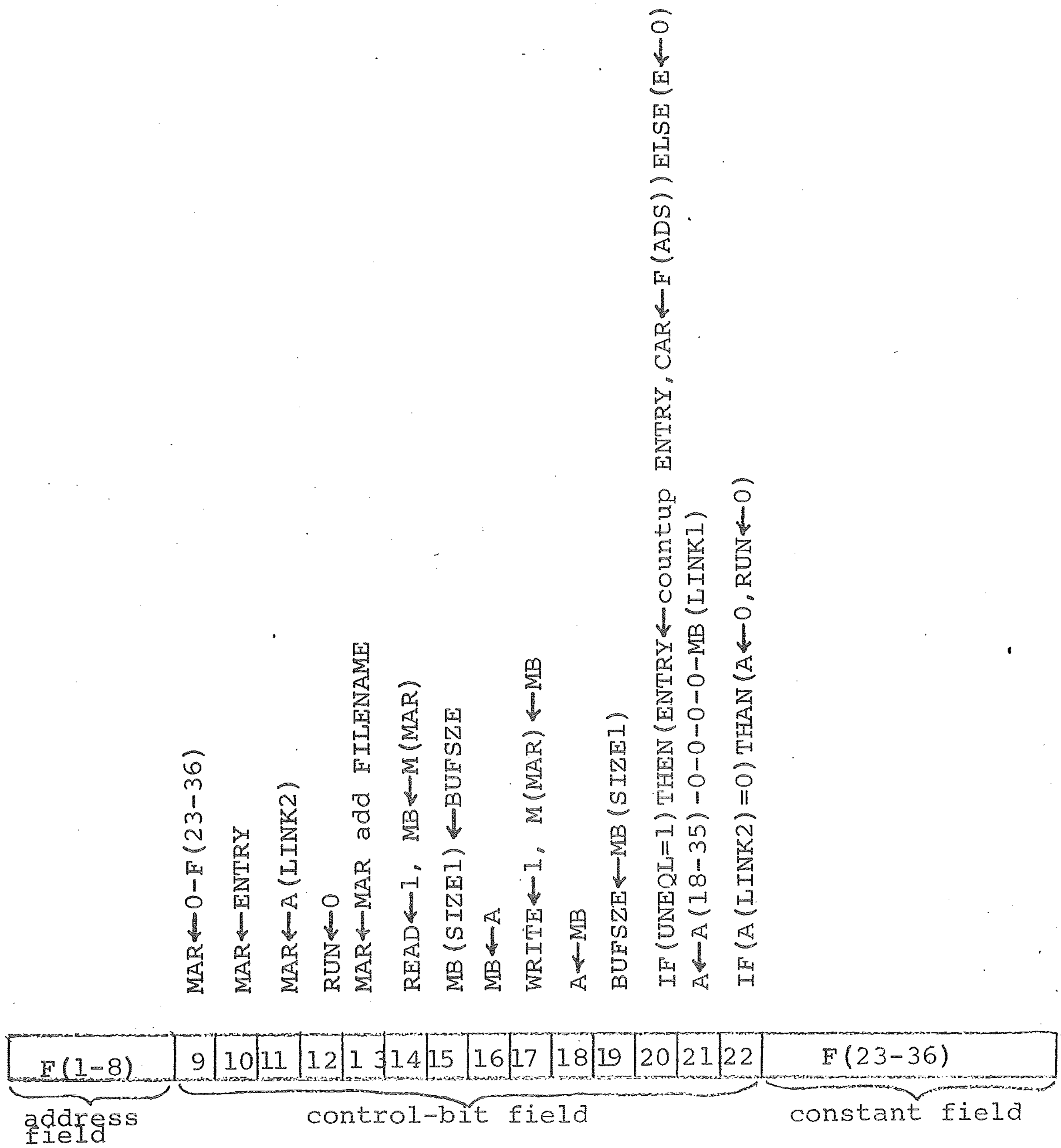

Fig.19 Control Word Format

Comment, execute the micro-instruction a t location 0.

```
/MC(0)*P(1)*RUN*F(13)/     MAR<-MAR add FILENAME,
/MC(0)*P(1)*RUN*F(14)/     READ<-1,
/MC(1)*P(1)*RUN*F(14)/     MB<-M(MAR),
/MC(2)*P(1)*RUN*F(19)/     BUFSZE<-MB(SIZE1),
/MC(3)*P(2)*RUN/           E<-0,
```

Comment, fetch the micro-instruction at location 1.

```
/P(3)*RUN*E'/              CAR<-countup CAR,
/P(0)*RUN*E'/              F<-CM(CAR),E<-1,
```

Comment, execute the micro-instruction at location 1.

```
/MC(0)*P(1)*RUN*F(10)/     MAR<-ENTRY,
/MC(0)*P(1)*RUN*F(14)/     READ<-1,
/MC(1)*P(1)*RUN*F(14)/     MB<-M(MAR),
/MC(1)*P(3)*RUN*F(18)/     A<-MB,
/MC(2)*P(1)*RUN*F(20)/     IF(UNEQL=1) THEN (ENTRY<-countup ENTRY,CAR<-0)
                                 ELSE (E<-0),
/MC(3)*P(2)*RUN/           E<-0,
```

Comment, initiate and fetch the micro-instruction at location 2.

```
/P(3)*RUN*E'/              CAR<-countup CAR,
/P(0)*RUN*E'/              R<-CM(CAR),E<-1,
```

Comment, execute the micro-instruction at location 2.

```
/MC(3)*P(1)*RUN*F(22)/     IF(A(LINK2)=0) THEN (A<-0,RUN<-0)
/MC(3)*P(2)*RUN/           E<-0,
```

Comment, initiate and fetch the micro-instruction at location 3.

```
/P(3)*RUN*E'/              CAR<-countup CAR,
/P(0)*RUN*E'/              F<-CM(CAR),E<-1
```

Comment, execute the micro-instruction at location 3.

```
/MC(0)*P(1)*RUN*F(11)/     MAR<-A(LINK2)
```

```
/MC(0)*P(1)*RUN*F(14)/     READ<--1,

/MC(1)*P(1)*RUN*F(14)/     MB<--M(MAR),

/MC(2)*P(2)*RUN*F(21)/     A<--A(18-350)-0-0-0-0-MB(LINK1),

/MC(3)*P(1)*RUN*F(22)/     IF (A(LINK2)=0)THEN (A<--0,RUN<--0),

/MC(3)*P(2)*RUN*/          E<--0,

Comment, initiate and fetch the micro-instruction at location 4.

/P(3)*RUN*E'/              CAR<--countup CAR,

/P(0)*RUN*E'/              F<--CM(CAR),E<--1,

Comment, execute the micro-instruction at location 4.

/MC(0)*P(1)*RUN*F(10)/     MAR<--ENTRY,

/MC(0)*P(1)*RUN*F(17)/     WRITE<--1,

/MC(0)*P(3)*RUN*F(15)/     MB(SIZE1)<--BUFSZE,

/MC(2)*P(1)*RUN*F(17)/     M(MAR)<--MB,

/MC(3)*P(1)*RUN*F(9)/      MAR<--0-F(23-36),

/MC(3)*P(2)*RUN/           E<--0,

Comment, initiate and fetch the micro-instruction at location 5.

/P(3)*RUN*E'/              CAR<--countupCAR,

/P(0)*RUN*E'/              F<--CM(CAR),E<--1,

Comment, execute the micro-instruction at location 5.

/MC(0)*P(1)*RUN*F(13)/     MAR<--MAR add FILENAME,

/MC(0)*P(1)*RUN*F(17)/     WRITE<--1,

/MC(0)*P(3)*RUN*F(16)/     MB<--A,

/MC(2)*P(1)*RUN*F(17)/     M(MAR)<--MB

/MC(3)*P(1)*RUN*F(12)/     RUN<--0

                           END
```

In the above, the initialization of all micro-operations is carried out by the START switch. There are six micro-instructions. The beginning of each micro-instruction, there are several execution statements. The first execution statement describes the fetch of the micro-instruction and the remaining statements describe the micro-operations that are executed by the micro-instruction. The fetch of the micro-instruction is initiated by setting register E to 0 and the execution of the micro-instruction is initiated by setting register E to 1. Register E is set to 1 during the first step of the execution of the micro-instruction, and it is reset to 0 by the START switch or during the last step.

In the above description, the braching of the loop in the sequence chart is accomplished by the following statement (whose control bit is in location 1),

```
/MC(2)*P(1)*RUN*F(20)/     IF(UNEQL=1) THEN(ENTRY<-countup ENTRY, CAR<-0)
                                       ELSE(E<-0),
```

If terminal UNEQL is equal to 1, then the following sequence occurs,

```
/MC(3)*P(2)*RUN/           E<-0,
/P(3)*RUN*E'/              CAR<-countup CAR,
/P(0)*RUN*E'/              F<-CM(CAR), E<-1,
```

Since terminal UNEQL is equal to 1, register CAR is reset to 0 and later again incremented by 1. The next micro-instruction is fetched at location 1. Therefore, the sequence is branched back to the beginning of the loop.

If terminal UNEQL is not equal to 1, then the following sequence occurs,

```
/P(3)*RUN*E'/              CAR<-countup CAR,
/P(0)*RUN*E'/              F<-CM(CAR), E<-1,
```

Since terminal UNEQL is not equal to 1, register CAR is incremented by 1. The next micro-instruction is fetched at location 2 because the contents of register CAR were 1. In this way, the micro-instruction exits from the loop.

4.9 Microprogram

The microprogram for the buffer allocation is shown in Figs. 20 and 21. The microprogram in Fig. 20 shows the micro-operations for each control bit, while that in Fig. 21 shows the 6 micro-instructions in octal. Each micro-instruction specifies the execution of those micro-operations whose control bits are 1. For example, the first micro-instruction has 1 in control bits 13, 14, and 19. Therefore, when this micro-instruction is executed, the following micro-operations at the specified control signals are performed,

/MC(0)*P(1)*RUN*F(13)/	MAR←-MAR add FILENAME,	(7)
/MC(0)*P(1)*RUN*F(14)/	READ←-1,	
/MC(1)*P(1)*RUN*F(14)/	MB←-M(CAR),	
/MC(2)*P(1)*RUN*F(19)/	BUFSZE←-MB(SIZE1),	

These are the previously described execution statements in the micro-instruction located at address 0, except the one which resets register E to 0, which occurs as the last step of each main memory cycle. Thus, the 1's and 0's in each previously described micro-instruction is translated into 1's and 0's. This is the way that the microprogram in Figs. 20 and 21 is obtained.

	Mc(3)	MC(0)	MC(0)	MC(3)	MC(0)	MC(0) / MC(1)	MC(0)	MC(0)	MC(0) / MC(2)	MC(1)	MC(2)	MC(2)	MC(2)	MC(3)	
	P(1)	P(1)	P(1)	P(1)	P(1)	P(1) / P(1)	P(3)	P(3)	P(1) / P(1)	P(3)	P(1)	P(1)	P(2)	P(1)	
	MAR←0-F(23-36)	MAR←ENTRY	MAR←A(LINK2)	RUN←0	MAR← MAR add FILENAME	READ←1 / MB←M(MAR)	MB(SIZE1)←BUFSZE	MB←A	WRITE←1 / M(MAR)←MB	A←MB	BUFSZE←MB(SIZE1)	IF(UNEQL=1) THEN (CAR←F(ADS), ENTRY←countup ENTRY) ELSE (E←0)	A←A(18-35)-0-0-0-MB(LINK1)	IF(A(LINK2)=0) THEN (A←0, RUN←0)	
F(1-8)	9	10	11	12	13	14	15	16	17	18	19	20	21	22	F(23-36)
0 0 0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0 0 0 0 0
0 0 0	0	1	0	0	0	1	0	0	0	1	0	1	0	0	0 0 0 0 0
0 0 0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0 0 0 0 0
0 0 0	0	0	1	0	0	1	0	0	0	0	0	0	1	1	0 0 0 0 0
0 0 0	1	1	0	0	0	0	1	0	1	0	0	0	0	0	0 0 0 0 2
0 0 0	0	0	0	1	1	0	0	1	1	0	0	0	0	0	0 0 0 0 0

Fig.20 The microprogram for the buffer allocation

Control Memory Address	Micro-instruction (Octal number)
0	0 0 0 0 6 0 4 0 0 0 0 0
1	0 0 0 4 2 1 2 0 0 0 0 0
2	0 0 0 0 0 0 0 4 0 0 0 0
3	0 0 0 2 2 0 1 4 0 0 0 0
4	0 0 1 4 1 2 0 0 0 0 0 2
5	0 0 0 1 4 6 0 0 0 0 0 0

Fig.21 The microprogram for the buffer allocation

5. Translation of Relocatable Code to Executable Code

Most contemporary digital computers are designed to execute machine instructions with absolute addresses (i.e., hardware addresses). A program written in such machine instructions is called an executable code. However, the computer user of today writes the program either in a symbolic assembly language or in a symbolic procedural language. Part of the task of translating a symbolic code to an executable code is chosen as the third example.

The translation task normally occurs in two steps. In the first step, the assembler (in the case of an assembly language) or the compiler (in the case of a procedural language) translates the program into an intermediate form, so that the user's program can be linked with other subprograms to form one program. The program in the intermediate form is a relocatable code because the program contains information which allows it to be located at another absolute address.

The second step consists of three functions: (a) assemblying one or more relocatable elements into a complete program, (b) assigning each subprogram an absolute address, and (c) translating the relocatable code into an executable code (i.e., changing all relative addresses to absolute addresses). These three functions are often carried out by a program commonly called a loader.

This example implements the third function of translating relocatable code to executable code by means of a microprogram.

5.1 The Input and Output

The input to the loader is one or more relocatable elements.

Each relocatable element consists of two parts, (a) relocatable code and (b) symbolic address tables. As mentioned, the relocatable code is the intermediate form of the machine language instructions and data that result from the translation of an assembly language subprogram or of a procedural language subprogram. A machine language instruction is usually made up of a non-address part (e.g., the op-code part) and an address part. Since relocatable translation requires only the adjustment of addresses, it is only necessary to distinguish between the address part and the non-address part of the instructions. Therefore, a relocatable code consists of a sequence of relocatable words. Each relocatable word contains an address part or a non-address part of an instruction. In this context, a data word can be viewed as an instruction without an address part.

Figure 22 shows the five formats (called A,B,C,D, and E) of the relocatable words. Each format has up to five fields: OP, FS, II, FLD, and INC. The FLD field contains data, or the address part of an instruction, or an index to a table. The FS field contains the length of the FLD field in bytes (for convenience, a byte is defined here as 3 bits). The OP field identifies the FLD field as,

(a) data (OP=1)

(b) Relative Address (OP=2) This address references a location within the subprogram relative to the subprogram address. At this address, the executable code is to be loaded.

(c) Common Data Address (OP=3) This address references a data area which is common to several subprograms.

(d) External Address (OP=4) This address references an entry point in other subprograms.

OP	FS	II	FLD	
1	length of FLD field (in bytes)		data part of instruction or data word	
3	4	2	3*(FS)	bits

(a) Format A, indicating data

OP	FS	II	FLD	
2	length of FLD field (in bytes)		address relative to the subprogram	
3	4	2	3*(FS)	bits

(b) Format B, indicating relative address

OP	FS	II	FLD	INC	
3	length of FLD field (in bytes)	incre-ment ind.	index to address in Common Symbol Table	possible increment or decrement	
3	4	2	3*(FS)	3*(FS)	bits

(c) Format C, indicating common data address

OP	FS	II	FLD	INC	
4	length of FLD field (in bytes)	incre-ment ind.	index to address in Undefined Symbol Table	possible increment or decrement	
3	4	2	3*(FS)	3*(FS)	bits

(d) Format D, indicating external address

OP	
5	
3	bits

(e) Format E, indicating the end of relocatable code

Figure 22 Relocatable Word Formats

(e) End of relocatable code (OP=5)

The above common data address and external address reference locations outside of the subprogram. These addresses are symbolic and are stored in the symbolic address tables to be described subsequently. The II and INC fields are provided to give a numerical increment to these symbolic addresses. If II is equal to 1 or 2, the address is incremented or decremented, respectively by the contents of INC. If II is equal to 0, there is no INC field.

Figure 23 shows an example of a relocatable code where the fields are separated by vertical lines and the numbers are octal. Words 1,2,3,4,6,8,9,11,12,13,15, and 17 are relocatable words with data (OP=1). Words 5,7,10, and 14 are those with relative addresses (OP=2). Word 16 is one with a common data address. Word 18 is the one with an external address. Word 19 is the one indicating the end of the relocatable code. Note that the relocatable words in Figure 23 are of different lengths. Though they are shown as left-justified, they are actually a string of bytes as shown in Figure 24. It is in the format of Figure 24 that the relocatable words are stored in the memory.

Symbolic address tables of a relocatable element contain all the symbolic addresses that are required to link subprograms together. There are three symbolic address tables:

(a) Defined Symbol Table (DST) This table contains the symbolic name of each entry point in the subprogram and its corresponding relative address in the subprogram.

(b) Undefined Symbol Table (UST) This table contains the symbolic

word \ byte	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Format Type
1	1	14	0	0	0	0	0	0	0	0	0	0	0	0	1	A
2	1	14	0	0	0	0	0	0	0	0	0	0	0	0	3	A
3	1	14	0	0	0	0	0	0	0	0	0	0	0	0	0	A
4	1	7	0	0	5	0	0	0	0	0						A
5	2	5	0	0	0	1	0	2								B
6	1	7	0	0	4	0	2	0	0	0						A
7	2	5	0	0	0	1	0	0								B
8	1	14	0	0	1	3	1	0	0	0	0	0	0	0	0	A
9	1	7	0	0	2	0	0	0	0	0						A
10	2	5	0	0	0	1	0	1								B
11	1	14	0	0	1	3	1	0	0	0	0	0	0	0	0	A
12	1	14	0	0	7	3	4	0	0	3	0	0	0	0	0	A
13	1	7	0	0	7	7	4	0	0	4						A
14	2	5	0	0	0	1	1	3								B
15	1	1	0	2												A
16	3	5	1	0	0	0	1	0	0	0	4	5	4			C
17	1	1	0	3												A
18	4	5	0	0	0	0	2	0								D
19	5															E
	OPR	FS	II	FLD					INC							

Figure 23 Example of a Relocatable Code (in octal)

‖ 1	6	0	0	0	0	0	0	0	0	0	0	1
0	0	1	‖ 1	6	0	0	0	0	0	0	0	2
0	0	0	0	0	3	‖ 1	6	0	0	0	0	3
0	0	0	0	0	0	0	0	0	‖ 1	3	4	4
0	5	0	0	0	0	0	‖ 2	2	4	0	0	5
1	0	2	‖ 1	3	4	0	4	0	2	0	0	6
0	‖ 2	2	4	0	0	1	0	0	‖ 1	6	0	7
0	1	3	1	0	0	0	0	0	0	0	0	8
1	3	4	0	2	0	0	0	0	0	‖ 2	2	9
4	0	0	1	0	1	‖ 1	6	0	0	1	3	10
1	0	0	0	0	0	0	0	0	‖ 1	6	0	11
0	7	3	4	0	0	3	0	0	0	0	0	12
1	3	4	0	7	7	4	0	0	4	‖ 2	2	13
4	0	0	1	1	3	‖ 1	0	4	2	‖ 3	2	14
5	0	0	0	1	0	0	0	4	5	4	‖ 1	15
0	4	3	‖ 4	2	4	0	0	0	2	0	‖ 5	16

one memory word
(36 bits)

Figure 24 Example of a Relocatable Code in the Memory (Double lines separate relocatable words and all numbers are octal.)

name of each external address that appeared in the subprogram. Each external address (OP=4) contains a link to an entry in this table.

(c) Common Symbol Table (CST) This table contains the symbolic name of each common data area reference in the subprogram. Each common data address (OP=3) contains a link to an entry in this table.

The formats of these tables in the relocatable element are not relevant here and are thus not shown. However, an example of these three tables as they appear in memory is shown in Figure 25; these tables will be further referenced.

The output from the loader is a sequence of machine instructions and data ready for loading into the memory. In order to properly locate the sequence in the memory, the sequence is prefaced by one word that contains the subprogram address.

Figure 26 shows an example of executable code. It is the output from the translation of the relocatable code in Figure 24. The instruction and data formats in Figure 26 follow those of IBM 7090/7094 computers. As mentioned, the first word holds the subprogram address (17000_8). It is assumed that the executable code is a part of a larger program and is stored in words 65 through 75. As shown, this example translates the 19 relocatable words into 12 machine instructions because some of the machine instructions consist of both address and data parts which are described by more than one relocatable word. However, the contents of the FLD fields of relocatable words 1,2,3,8,11, and 12 in Figure 23 correspond with words 65,66,67, 70,72, and 73, respectively, in Figure 26, because these data and

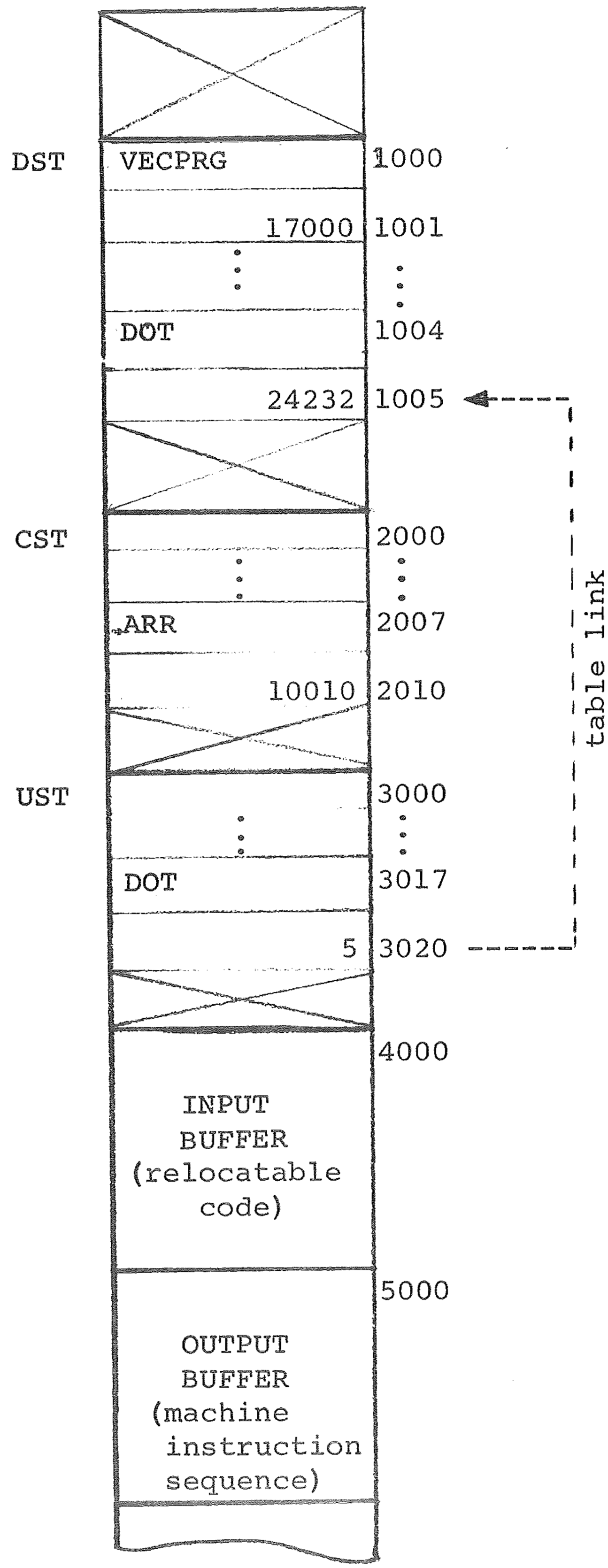

Figure 25   Example of Tables and Buffers in the Memory

Word 1	0	0	0	0	0	0	0	1	7	0	0	0
⋮						⋮						
Word 65	0	0	0	0	0	0	0	0	0	0	0	1
Word 66	0	0	0	0	0	0	0	0	0	0	0	3
Word 67	0	0	0	0	0	0	0	0	0	0	0	0
Word 68	0	5	0	0	0	0	0	1	7	1	0	2
Word 69	0	4	0	2	0	0	0	1	7	1	0	0
Word 70	0	1	3	1	0	0	0	0	0	0	0	0
Word 71	0	2	0	0	0	0	0	1	7	1	0	1
Word 72	0	1	3	1	0	0	0	0	0	0	0	0
Word 73	0	7	3	4	0	0	3	0	0	0	0	0
Word 74	0	7	7	4	0	0	4	1	7	1	1	3
Word 75	2	1	0	4	6	4	3	2	4	2	3	2

Figure 26 Example of an Executable Code

instruction words have no addresses.

5.2 Algorithm

The loader performs the translation. (For simplicity, it is assumed that the loader does not handle memory overlay.) The translation can be described in three phases. In the first phase, the relocatable elements are collected and references between subprograms (which include common data references, subprogram entry points, and external references) are tabulated. In the second phase, the external references are matched with their respective entry points, and each common data area, subprogram, and entry point are then assigned an absolute address. In the third phase, the relocatable code of each subprogram is translated to executable code by assigning each relative address an absolute address. The translation process to be described here is limited to the third phase.

The translation first unpacks the relocatable code (see the example in Figures 23 and 24) stored in the input buffer, then interprets the op-field of each word of the relocatable code, and finally places the data or the modified address assembled into a sequence of machine-language instructions in the output buffer. The translation process is shown in the flow chart in Figure 27. The relocatable elements are in the input buffer, and the machine-language-instruction sequence will be in the output buffer. Both input and output buffers are indicated in Figure 25. As shown in Figure 27, the initial step places the subprogram address in the output buffer. The first or next relocatable word is read out of the input buffer for unpacking. The unpacking process recognizes the beginning and the end of the relocatable word as well as the fields of the word. The OP field is first

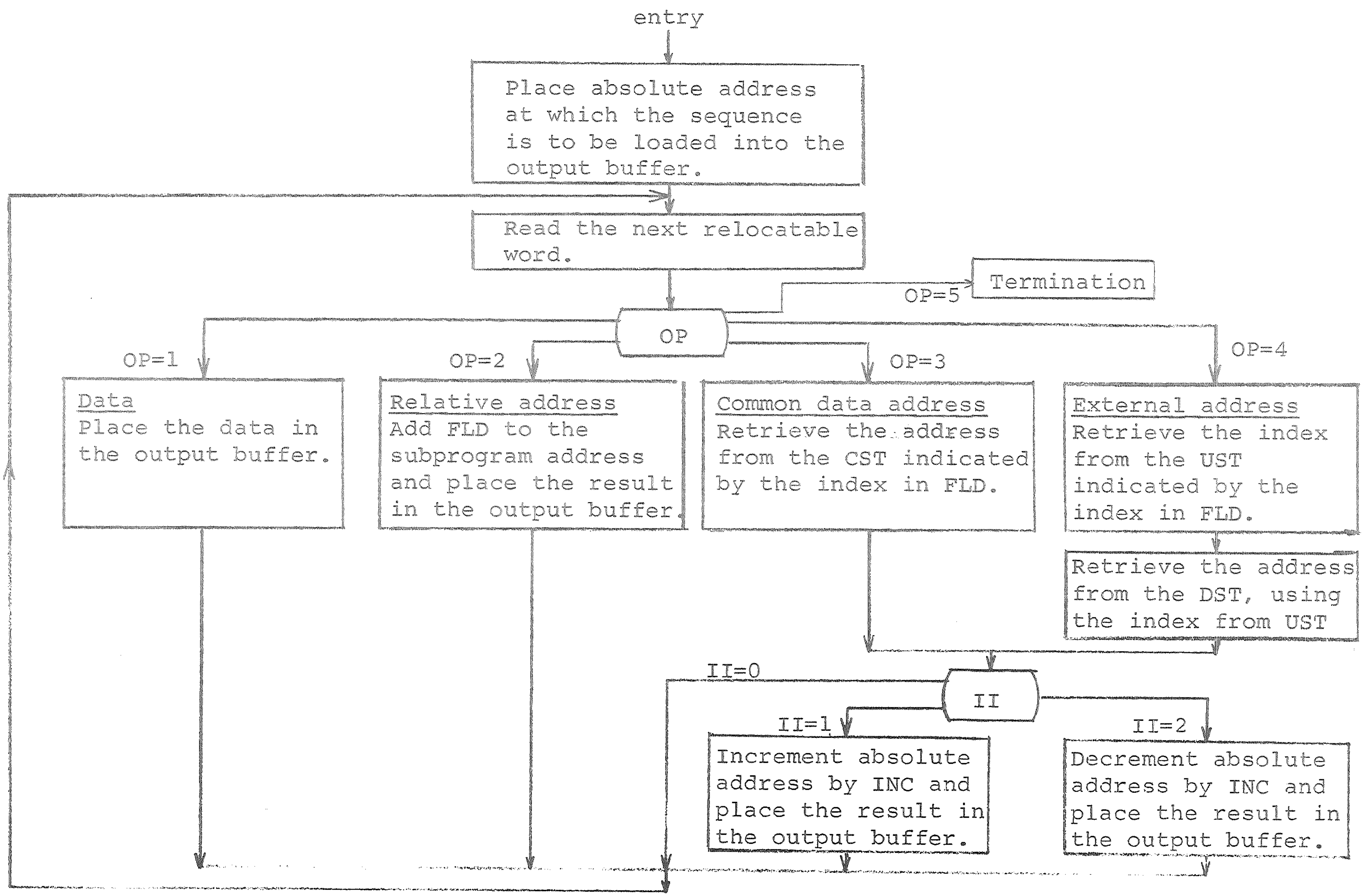

Figure 27 Flow chart showing the translation of relocatable code to executable code

decoded and the following address modification occurs.

(a) If OP is 1, it indicates data (format A shown in Figure 22). Since it is not an address, no address modification is required. The FS bytes of the FLD field are placed in the next available bytes in the output buffer, where FS denotes the size of the FLD field in bytes.

(b) If OP is 2, it indicates a relative address (format B). The FS bytes of the FLD field (the relative address) is added to the subprogram address (such as 17000_8 in Figure 26). The sum is placed in the next FS bytes of the output buffer.

(c) If OP is 3, it indicates a common data address (format C). The index in the FLD field (which is an address relative to CST) (such as 2000_8 in Figure 25). The absolute address stored at this location is retrieved. If field II contains 1 or 2, the contents of the INC are added to or subtracted from the absolute address, respectively. If field II contains 0, there is no address modification. In any of the three cases, the resulting address is then placed in the next FS bytes of the output buffer.

(d) If OP is 4, it indicates an external address (format D). The FS bytes of the FLD field (which is an address relative to UST) is added to the address of the Undefined Symbol Table (UST) (such as 3000_8 in Figure 25). The index stored at this location is retrieved and added to the address of the Defined Symbol Table (DST) (such as 1000_8 in Figure 25). Then, the absolute address at this location is retrieved. If field II is 0, no further modification of the address is required. If field II is 1 or 2, the contents of the INC field are added to or subtracted from the absolute address, respectively. In any of the three cases, the absolute address is placed in the next FS bytes of the output buffer.

(e) If OP is 5, it indicates the end of the relocatable code (format E). At this point, the translation is terminated.

After one of the above operations is performed, the next relocatable word is read out of the input buffer . Unpacking, decoding and address modification continues on until the end of the relocatable code. At this time, the translation is completed. This tranlation process will be further described in more detail later when the sequence charts are presented.

As an example, let the relocatable code in Figures 23 and 24 be the input; the output from the translation is the executable code shown in Figure 26. Word 1 in Figure 26 contains the absolute address 17000_8 at which the subsequent machine-language-instruction sequence is to be loaded. Words 2 through 64 are assumed to be some other part of the subprogram. Relocatable words 1, 2, and 3 in Figure 23 contain data and are thus translated without modification to words 65, 66, and 67 of the output buffer as shown in Figure 26. Relocatable word 4 in Figure 23 also contains data and is translated without modification to the first 7 bytes of word 68 in Figure 26. Relocatable word 5 in Figure 23 stores a relative address; thus, the contents of the FLD field are added to the subprogram address (17000_8) and the result is then placed as the last 5 bytes of word 68 (17102_8). Words 69 through 74 in Figure 26 are similarly translated from relocatable words 6 through 14 in Figure 23. Word 75 in Figure 26 is a machine instruction with two addresses; it is translated from relocatable words 15 through 18 in Figure 23. Relocatable word 15, which contains the op-code of the instruction, becomes the first byte of word 75. Relocatable word 16 contains an index (00010_8) to the Common Symbol Table (CST). At the 8th location relative to address CST (2000_8) in Figure 25, the absolute address is found to be 10010_8 whose symbolic address name is ARR. Since field II is 1, address 10010_8 is incremented by the contents of field INC to become bytes 2 through 6 of word 75 of the output buffer (10464_8). Relocatable word 17, which contains the index of the instruction, is translated without modification to byte 7. Relocatable word 18 contains an index (00020_8) to the Undefined Symbol Table (UST). At the 16th location relative to address UST (3000_8) as shown in Figure 25, the index is found to be 5 and symbolic name to be DOT. This index is the address (0005_8) relative to the location DST (1000_8) of the Defined Symbol Table. At this location (1005_8), absolute address 24232_8 is found. Since field II contains 0, no address modification is required. This absolute address is entry point DOT.

5.3 Configuration

The configuration of the microprogrammed loader is shown in Figure 28 except for the control part which is to be shown subsequently. Main memory M has address register AR and storage register SR. Registers READ and WRITE are used to initiate a memory read or memory write, respectively. The relocatable elements and the input and output buffers are stored in the memory. There are eight index registers, X1, X2,...X8, which store the table and buffer addresses during translation. Registers OP, FS, and II store the OP field, FS field, and the II field, respectively, of a relocatable word. Unpacking of a relocatable word and

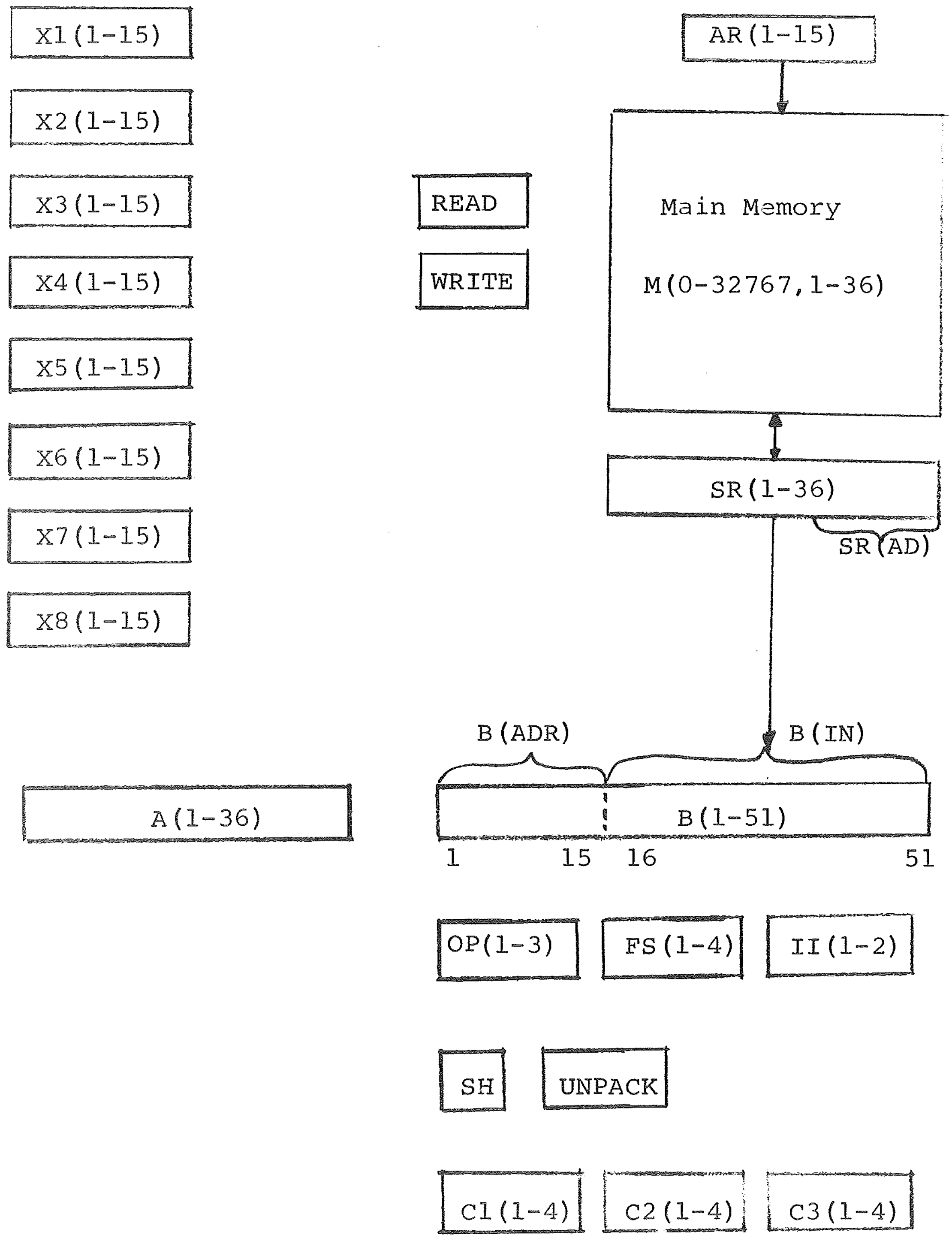

Figure 28 Configuration for Translating Relocatable Code

address modification of its address part are performed in registers A and B. Single-bit register SH indicates that register B or cas-register A-B is shifted to the left according to register SH containing a 0 or 1, respectively. Single-bit register UNPACK is a control register for calling the unpacking sequence. In addition, there are three counters C1, C2, and C3.

```
Comment, configuration of the translator                        (8)
Register,   A(1-36),             $accumulator
            B(1-51),             $unpacking register
            AR(1-15),            $address register
            SR(1-36),            $storage register
            X1(1-15),            $store the INC field
            X2(1-15),            $store INPUT address
            X3(1-15),            $store OUTPUT address
            X4(1-15),            $store CST
            X5(1-15),            $store UST
            X6(1-15),            $store DST
            X7(1-15),            $store subprogram address
            X8(1-15),            $temporary storage
            OP(1-3),             $op-register
            FS(1-4),             $field size register
            II(1-2),             $incrementing indicator
            C1(1-4),             $count left shifts in casregister AB
            C2(1-4),             $count leftshifts in register B
            C3(1-4),             $count leftshifts in register A
            SH,                  $shift-control register
            UNPACK,              $control register
```

	READ,	$memory read register
	WRITE,	$memory write register
Subregister,	B(ADR)=B(1-15),	$address part of unpacking register
	B(IN)=B(16-51),	$input part of unpacking register
	SR(AD)=SR(22-36)	$address part of storage register
Memory,	M(AR)=M(0-32791,1-36)	$main memory
Casregister,	AB(1-87)=A-B	

5.4 Sequence charts

The translation algorithm in Figure 27 is now converted into sequence charts. Block diagram in Figure 29 shows that the translation is organized into four sequences: initialization, fetch, address modification, and unpacking. The sequence charts are shown in Figures 30 to 33. As indicated by the dotted lines, the unpacking sequence is called during the fetch sequence and the address modification sequence. The initialization sequence initializes the translation. The fetch sequence fetches a relocatable word, unpacks it and decodes it. The address modification sequence performs the address modification. The unpacking sequence performs the task of reading a relocatable word out of the input buffer, shifting casregister AB to the left, and storing a machine instruction into the output buffer.

It is assumed that the relocatable element and the buffers are initially in the memory. The relocatable element is in the form of a string of digits as shown in Figure 24. The addresses of the tables and buffers are in the index registers as described below.

(a) input buffer location in register X2,

(b) output buffer location in register X3,

(c) CST location in register X4,

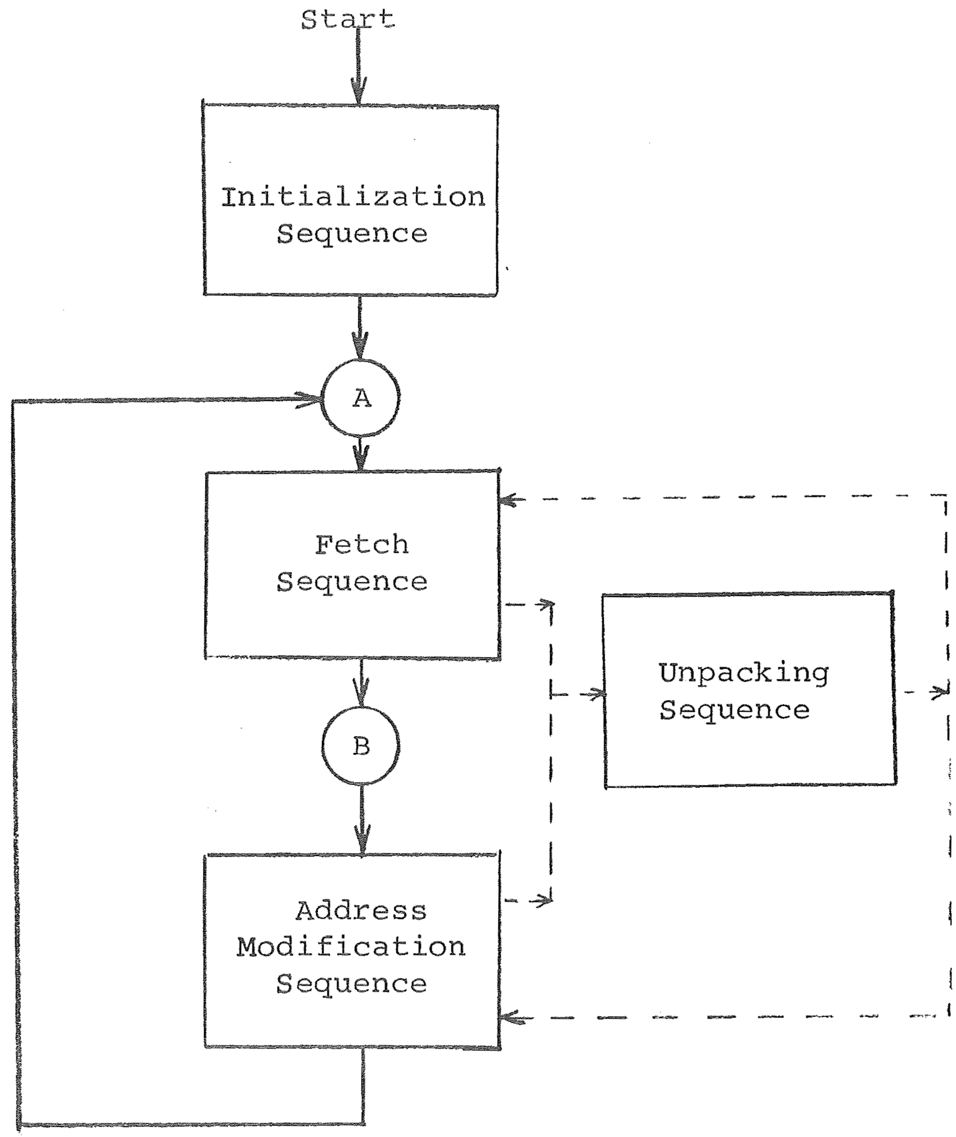

Figure 29 Flow chart showing the four sequences for translation of relocatable code to executable code

(d) UST location in register X5,

(e) DST location in register X6,

(f) subprogram address in register X7,

In the sequence charts, a memory read will be indicated with the CDL statement

SR<-M(AR).

A memory write will be indicated similarly as

M(AR)<-SR.

This is done for clarity alone. The actual read of memory is initiated by setting the register READ to one (READ--1) while the memory write is initiated by setting the register WRITE to one (WRITE--1). The transfer of information occurs as a result of this.

5.4.1 Initialization sequence

The initialization sequence as shown in Figure 30 performs five tasks. It reads the first word out of the input buffer (location in register X2) and stores it in subregister B(IN). It places the subprogram address in register X7 into subregister B(ADR). It increments register X2 by 1. It resets register A to 0. And it sets the initial contents of counter C1, C2, and C3 to be 5, 12, and 7, respectively.

Counter C1 counts the number of leftshifts of casregister AB in bytes. The shifting of the 5-byte subprogram address from subregister B(ADR) to subregister A(22-36) is controlled by setting counter C1 to 5 and then counting down until it reaches 0. Counter C2 counts the number of leftshifts of register B in bytes. The indication to read the next word from the input buffer into subregister

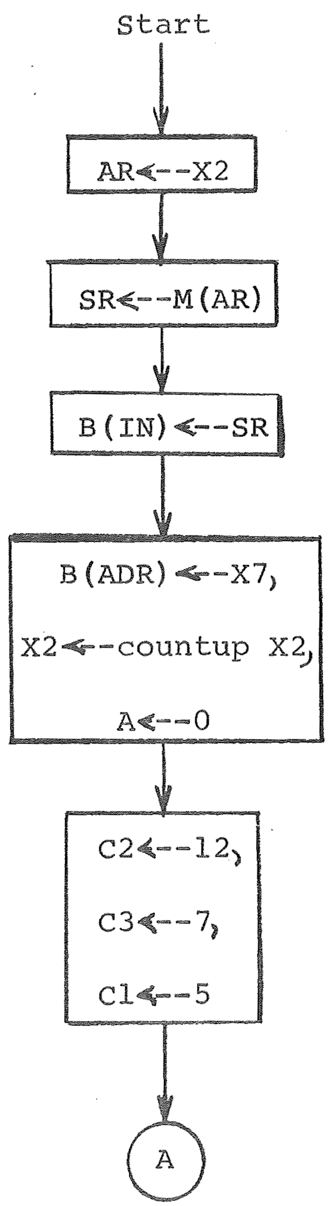

Figure 30 Sequence chart for the initialization sequence

B(IN) is given by setting counter C2 to 12 and then counting down until it reaches 0. Counter C3 counts the number of bytes that are shifted into register A where machine instruction is being assembled. The five leftshifts required to complete the first machine instruction in register A is controlled by setting counter C3 to 7 and then counting up until it reaches 12.

5.4.2 Fetch sequence

The fetch sequence as shown in Figure 31 performs four tasks. It shifts the word in subregister B(IN) the number of byte positions to the left indicated by counter C1 so that the next relocatable word is now left-adjusted in register B. By making this leftshift occuring in casregister AB, it also shifts the address or data in the left part of register B into register A. It then transfers the contents of OP, FS, and II fields in subregister B(1-3), B(4-7), and B(8-9) to registers OP, FS, and II, respectively. Since these three fields in subregister B(1-9) are of no further use, register B is leftshifted 3 byte positions so that the FLD field of the relocatable word is left-adjusted in register B.

In the above tasks, there are two left shifts; one in register B and the other in casregister AB. These two shifts are indicated by register SH containing 0 and 1, respectively. Such a leftshift is also required in the address modification sequence. For convenience, a subsequence called the unpacking subsequence is formed. This subsequence is "called" by setting register UNPACK to 1 and "returns" to the calling sequence by resetting register UNPACK to 0 in the subsequence. After the sequence register UNPACK is set to 1, the

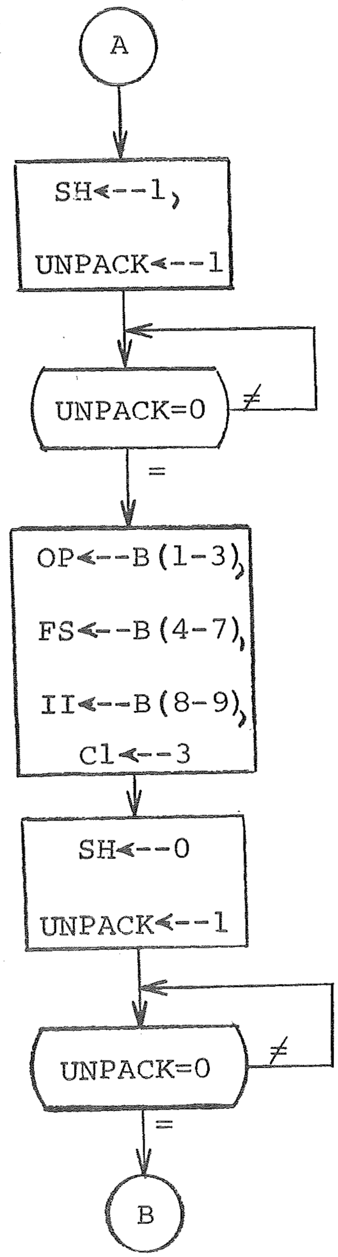

Figure 31 Sequence chart for the fetch sequence

sequence constantly examines register UNPACK and waits for its contents to become 0. When register UNPACK is being set to 1, register SH should also be set to 0 or 1 in order to select one of the two possible leftshifts.

5.4.3 Unpacking subsequence

The unpacking subsequence as shown in Figure 32 performs four tasks. The first task carries out the leftshift as described by the following conditional micro-statement.

IF (SH=0) THEN (B←3 shl B) ELSE (AB←3 shl AB)

and decrements counter C1 until it reaches 0. When counter C1 becomes 0, register UNPACK is reset to 0. The second task is to read a word out of the input buffer located by register X2 into subregister $B^{(IN)}$; this is controlled by counter C2. When counter C2 reaches 0, reading of the word from the input buffer is carried out. The third task is to store a machine instruction assembled in register A into the output buffer located by register X3; this is controlled by counter C3. When counter C3 reaches 12, storing of the assembled instruction in register A is carried out. This can logically occur only after C1 becomes zero. The fourth task is waiting. As shown in Figure 32, there is a waiting loop during which the UNPACK subsequence constantly examines register UNPACK and waits for its contents to become 0.

It should be noted that in order to call the UNPACK subsequence, register UNPACK should be set to 1, register SH should be set to 0 or 1, and counter C1 should be set to a certain initial value.

5.4.4 Address modification sequence

The address modification sequence as shown in Figure 33 performs the operations specified by the OP and II fields on the operands

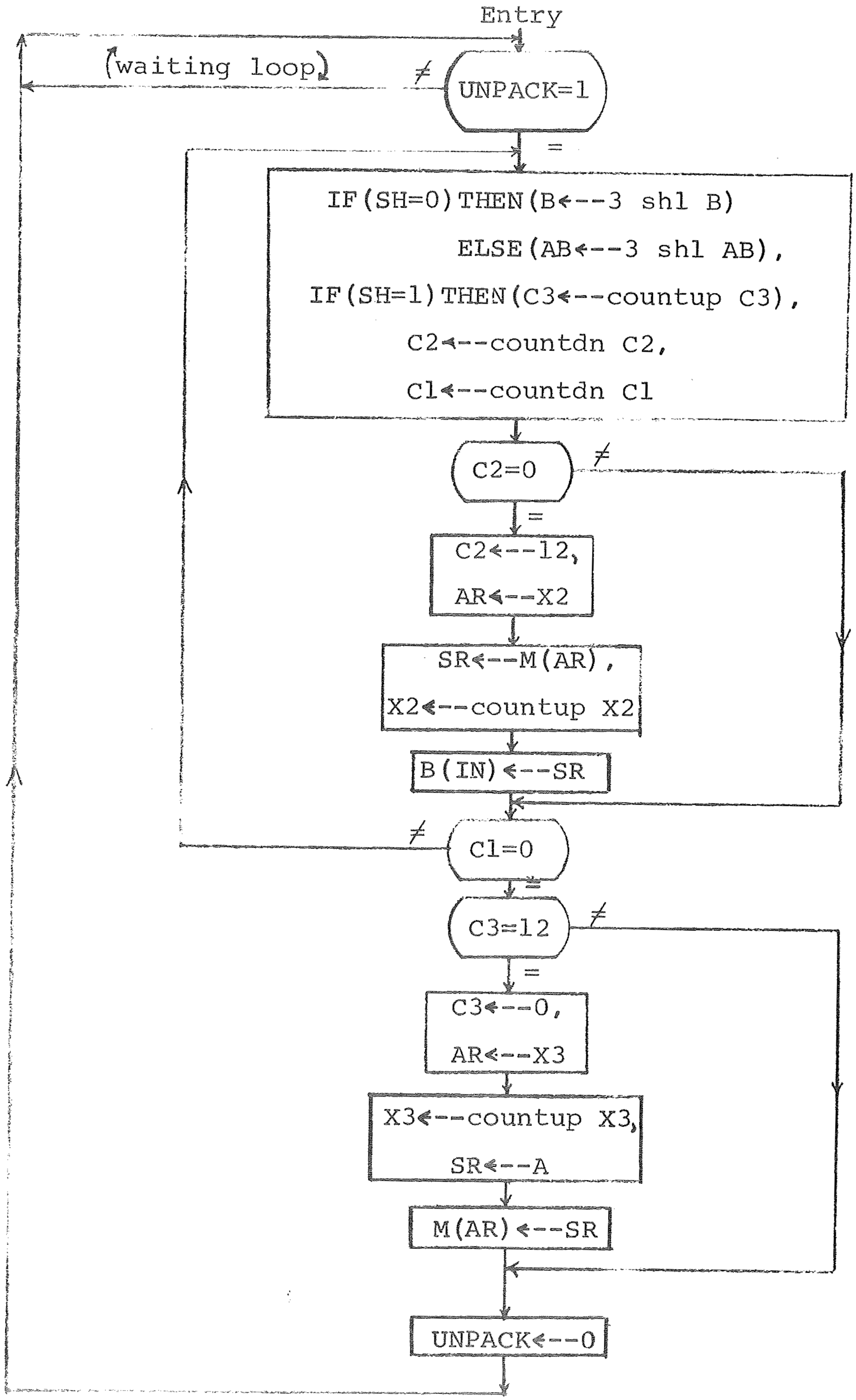

Figure 32 Sequence chart for the unpacking subsequence

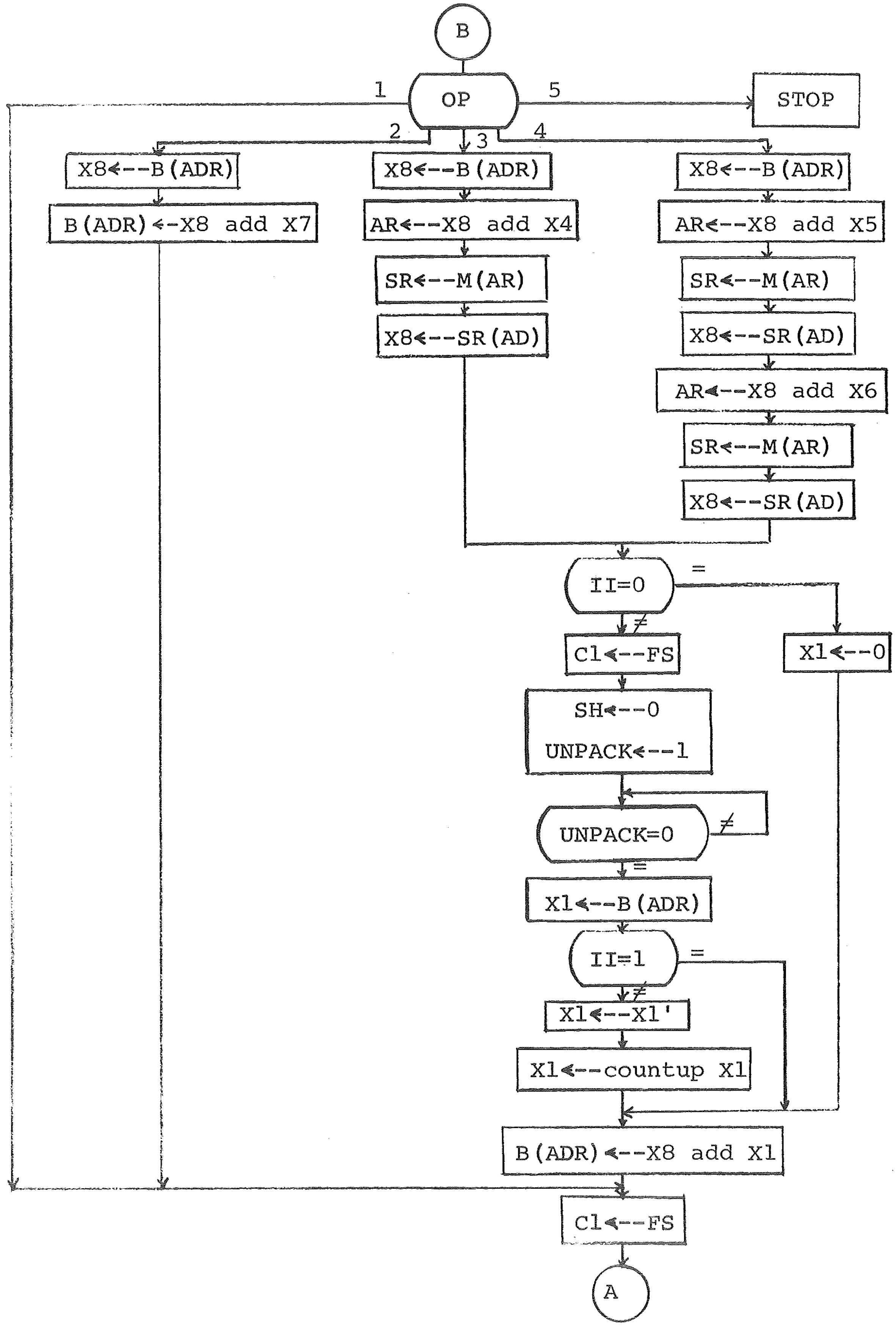

Figure 33 Sequence chart for the address modification sequence

in the FLD and INC fields. As shown in Figure 33, if the OP field contains 1, there is no address modification. If the OP field contains 2, the address in subregister B(ADR) is incremented by the subprogram address in register X7. Whether the OP field is 1 or 2, the contents of the FS field are transferred to counter C1.

If the OP field contains 3, the index in subregister B(ADR) is incremented by the location of the Common Symbol Table in register X4. The word is read out of this memory location and stored in register X8. If the OP field contains 4, the index in subregister B(ADR) is incremented by the location of the Undefined Symbol Table in register X5. At this location is another address. This address is read out of the memory, stored in register X8, and incremented by the location of the Defined Symbol Table in register X6. Then, the contents of this memory location are read out of the memory and stored in register X8.

If the OP field contains 3 or 4 and if the II field is not 0, an addition or a subtraction is yet required. The contents of the INC field are first shifted into subregister B(ADR) and are then added (if II is 1) to or subtracted (if II is 2) from the contents in register X8 with the result stored in subregister B(ADR). The subtraction is performed by addition of 2's complement of the subtrahend. Whether the OP field is 3 or 4, the contents of the FS field are transferred to counter C1.

At this point, the address modification sequence is completed and returns to the fetch sequence.

5.5 Microprogram control configuration

The block diagram in Figure 34 shows the configuration of the

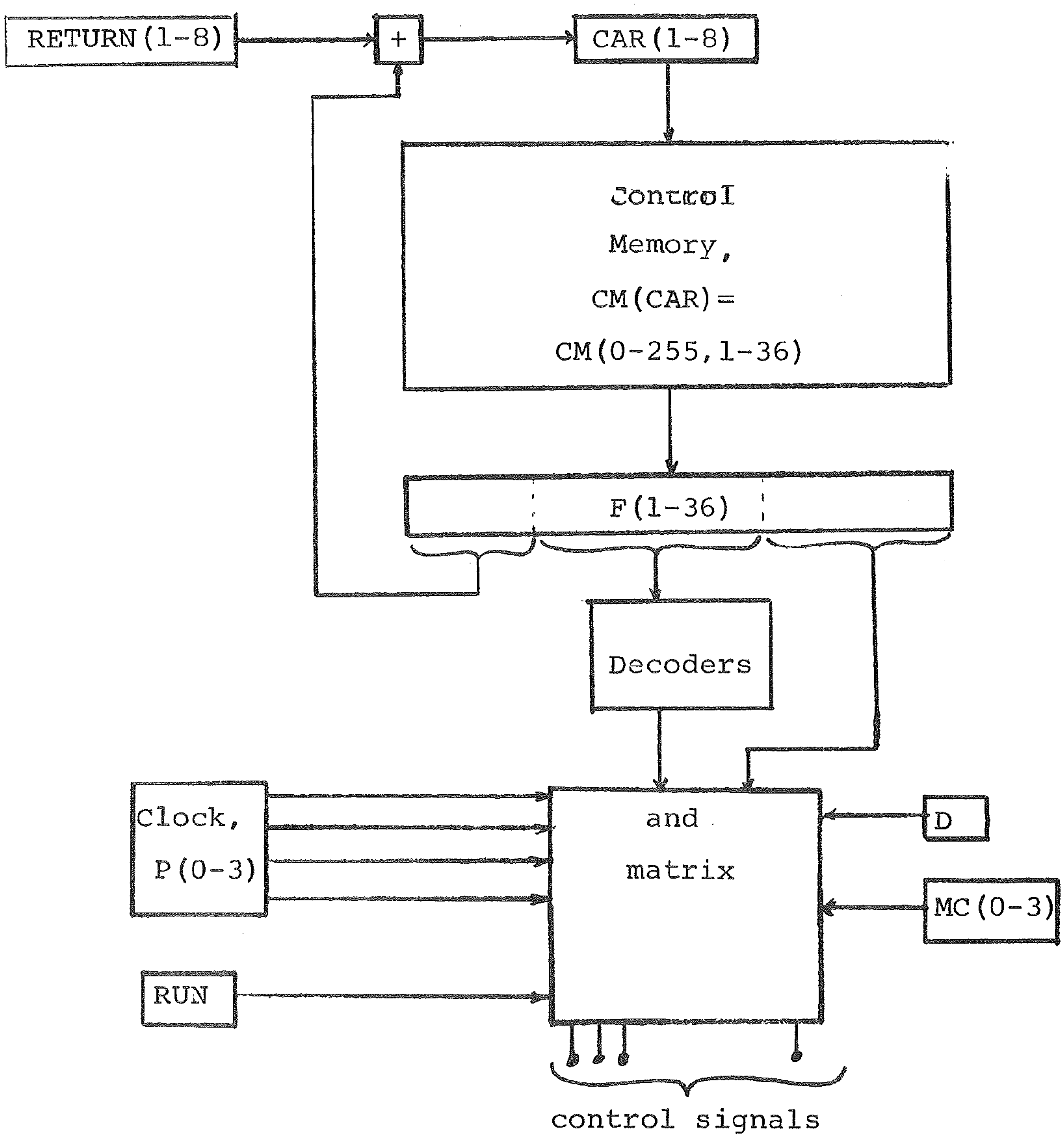

Figure 34 Block diagram showing the configuration of the control unit

control unit. Control memory CM has a capacity of 256 36-bit words with address register CAR and buffer register F. The 8-bit register RETURN stores a control memory address for subroutine return. The four-phase clock P(0-3) in conjunction with the single-bit registers RUN and C, and the 4-bit register MC generates the control signals. Switch START initiates the micro-programmed computer. The above configuration is now described by the following CDL statements.

```
Comment, configuration for the microprogram control                (9)
Register,      CAR(1-8),              $control memory address register
               F(1-36),               $control word register
               RETURN(1-8),           $micro-subroutine return register
               MC(0-3),               $register for sequencing main memory cycles
               D,                     $memory cycle wait register
               RUN,                   $start-stop register
Subregister,   F(ADS)=F(1-8)          $address portion of the control word
Memory,        CM(CAR)=CM(0-255,1-36),
Switch,        START(ON),             $start switch

Comment, each control memory cycle coincides with one clock cycle, and
         each main memory cycle coincides with four control memory cycles.
Clock,         P(0-3),                $four-phase clock
```

5.6 Timing and Control Signals

Each main memory cycle is chosen to consist of four control memory cycles, and each control memory cycle coincides with one clock cycle. Therefore, there are 4 steps in each control memory cycle and 16 steps in each main memory cycle. The control signals for these 16 steps are described by the following sequence of 16 labels,

```
Comment, control signals expressed by the labels                  (10)
/MC(0)*P(0)*RUN/             $beginning of a main and a control memory cycle
/MC(0)*P(1)*RUN/
/MC(0)*P(2)*RUN/
/MC(0)*P(3)*RUN/             $end of a control memory cycle
/MC(1)*P(0)*RUN/             $beginning of a control memory cycle
/MC(1)*P(1)*RUN/
/MC(1)*P(2)*RUN/
/MC(1)*P(3)*RUN/             $end of a control memory cycle
/MC(2)*P(0)*RUN/             $beginning of a control memory cycle
/MC(2)*P(1)*RUN/
/MC(2)*P(2)*RUN/
/MC(2)*P(3)*RUN/             $end of a control memory cycle
/MC(3)*P(0)*RUN/             $beginning of a control memory cycle
/MC(3)*P(1)*RUN/
/MC(3)*P(2)*RUN/
/MC(3)*P(3)*RUN/  D<-0,      $end of both memory cycles
```

In the above labels, the four steps in each control memory cycle are controlled by the four phases of clock P(0-3) and the four control memory cycles in each main memory cycle are controlled by the four states of ring counter MC(0-3). Register RUN is employed to activate the control signals for the 16 steps in a main memory cycle as indicated in Figure 34.

During each main memory cycle, an instruction is read out of or written into the main memory. It is now specified that the transfer of the main memory address to register AR and the initiation of the main memory read or write must occur during the second step (i.e., /MC(0)*P(1)*RUN/). For a read operation, the word is available at buffer register SR during the sixth step (i.e., /MC(1)*P(1)*RUN/). For a write operation, the word to be stored into the memory is transfered into buffer register SR before the 12th step (/MC(2)*P(3)*RUN/).

If certain micro-operations occur in every control memory cycle, the following sequence of four labels is used,

```
/P(0)*RUN*D'/    F<-CM(CAR)         $beginning of a control memory cycle
/P(1)*RUN*D'/
/P(2)*RUN*D'/
/P(3)*RUN*D'/                       $end of a control memory cycle
```

In the above sequence of labels, register D is used to control the advance or stop of the 4 steps in a control memory cycle. When register D contains a 0, the sequence of the labels exist; otherwise, it disappears.

During each control memory cycle, a micro-instruction is read out of the control memory. It is now specified that the transfer of the control memory address to register CAR and the initiation of the control memory read must occur during clock phase P(3) of the preceeding control memory cycle, and the control word becomes available at buffer register F during the first clock phase P(0) of the current control memory cycle. Micro-operations activated by the micro-instruction in register are executed during clock phases P(1-3) of the current control memory cycle.

Register D is automatically set to zero at the end of each main memory cycle (/MC(3)*P(3)*RUN/). Thus, when waiting to the beginning of the main memory cycle is required, register D is set to one to stop generation of the control signals during a control memory cycle, but the control signals for the main memory cycle continue. If a micro-instruction is fetched at the beginning of a main memory cycle and register D is set to one at the same time, then the micro-instruction remains in register F for one main memory cycle, as will be later described.

5.7 Control word format

Table 3 shows the format of the control word. The 36 bits of each control word in register F are divided into three groups:

(a) field F(1-8) which contains a control memory address,

(b) field F(9-23) which is divided into five subfields with a decoder attached to each subfield,

(c) field F(24-36) where each bit controls one micro-operation or a group of micro-operations.

There are 28 control bits in field F(9-36) which control 43 execution statements.

Field F(1-8) provides a two-way branch to each micro-instruction. The five subfields in field F(9-23) are: F(9-11), F(12-13), F(14-16), F(17-20), and F(21-23). Each subfield controls micro-operations whose occurrences are mutually exclusive. Field F(9-11) controls micro-operations which initialize the three counters. Field F(12-13) controls counting micro-operations. Field F(14-16) controls micro-operations which load address register AR. Field F(17-20) controls micro-operations which involve storage register SR and register B. Field F(21-23) controls micro-operations which set up the control memory address in

Table 3 Control Word Format

Control Bits	Decoder	Control Signal	Micro-operations
F(1-8)			Control memory address field
F(9-11)	DC(0-7)		
	DC(1)	D'*P(1)	C1<-FS,
	DC(2)	D'*P(1)	C1<-3,
	DC(3)	D'*P(1)	C1<-5,
	DC(4)	D'*P(1)	C2<-12,
	DC(5)	D'*P(1)	C3<-0,
	DC(6)	D'*P(1)	C3<-7,
F(12-13)	DX(0-3)		
	DX(1)	D'*P(2)	X1<-countup X1,
	DX(2)	D'*P(2)	X2<-countup X2,
	DX(3)	D'*P(2)	X3<-countup X3,

Table 3 Continued

Control Bits	Decoder	Control Signal	Micro-operations
F(14-16)	DAR(0-7)		
	DAR(1)	MC(0)*P(1)	AR<-X2,
	DAR(2)	MC(0)*P(1)	AR<-X3,
	DAR(3)	MC(0)*P(1)	AR<-X8 add X4,
	DAR(4)	MC(0)*P(1)	AR<-X8 add X5,
	DAR(5)	MC(0)*P(1)	AR<-X8 add X6,
F(17-20)	DBS(0-15)		
	DBS(1)	D'*P(2)	SR<-A,
	DBS(2)	MC(2)*P(1)	B(IN)<-SR,
	DBS(3)	MC(2)*P(1)	X8<-SR(AD),
	DBS(4)	D'*P(1)	X1<-B(ADR),
	DBS(5)	D'*P(1)	B(ADR)<-X7,
	DBS(6)	D'*P(1)	X8<-B(ADR),
	DBS(7)	D'*P(1)	B(ADR)<-X8 add X1,
	DBS(8)	D'*P(1)	B(ADR)<-X8 add X7,
F(21-23)	DT(0-7)		
	DT(1)	D'*P(3)	CAR<-countup CAR
	DT(2)	D'*P(3)	CAR<-F(ADS)
	DT(3)	D'*P(3)	CAR<-F(1-5)-OP
	DT(4)	D'*P(3)	CAR<-RETURN
	DT(5)	D'*P(3)	IF(II=0) THEN (CAR<-F(ADS)) ELSE (CAR<-countup CAR)
	DT(6)	D'*P(3)	IF(II=1) THEN (CAR<-F(ADS)) ELSE (CAR<-countup CAR)
	DT(7)	MC(2)*P(3)	CAR<-countup CAR
		MC(3)*P(3)	IF(C1≠0) THEN (CAR<-F(ADS)),
			IF ((C1=0)*(C3≠12)) THEN (CAR<-RETURN),

Table 3 Continued

Control Bits	Decoder	Control Signal	Micro-operations
F(24)		D'*P(1)	IF(SH=0) THEN (B<-3 shl B) ELSE (AB<-3 shl AB),
			IF(SH=1) THEN (C3<-countup C3),
			C1<-countdn C1,
			C2<-countdn C2,
		D'*P(3)	IF(C2=0) THEN(CAR<-countup CAR,
			IF (MC(0)+MC(1)+MC(2)=1) THEN (D<-1)),
			IF((C1=0)*(C2≠0)*(C3=12)) THEN (CAR<-F(ADS),
			IF (MC(0)+MC(1)+MC(2)=1) THEN (D<-1))
			IF((C1=0)*(C2≠0)*(C3≠12)) THEN (CAR<-RETURN)
F(25)		D'*P(1)	OP<-B(1-3), FS<-B(4-7), II<-B(8-9)
F(26)		D'*P(2)	RETURN<-CAR,
		D'*P(3)	RETURN<-countup RETURN
F(27)		D'*P(1)	A<-0,
F(28)		D'*P(1)	X1<-0
F(29)		D'*P(1)	X1<-X1'
F(30)			(not used)
F(31)		MC(0)*P(1)	READ<-1
F(32)		D'*P(2)	SH<-0
F(33)		D'*P(3)	IF(MC(0)+MC(1)+MC(2)=1) THEN (D<-1)
F(34)		MC(0)*P(1)	WRITE<-1
F(35)		D'*P(2)	SH<-1
F(36)		D'*P(3)	RUN<-0

register CAR.

The 13 control bits in field F(24-36) control the remaining micro-operations. Note that bit F(24) controls the shift and test micro-operations involving register B and casregister AB in two clock phases P(1) and P(3).

5.8 Statement description

This section presents the microprogram described in the CDL statements. The microprogram consists of 24 micro-instructions: 3 for the unpacking subsequence, 2 for the initialization sequence, 2 for the fetch sequence, and 16 for the address modification sequence.

5.8.1 Unpacking subsequence

The three micro-instructions for this subsequence are described below.

```
Comment, unpacking subsequence
Comment, shift-and-test micro-instruction located at C.M. address 63   (11)
/D'*RUN*P(0)/         F<-CM(CAR),
/D'*RUN*P(1)*F(24)/   IF (SH=0) THEN (B<-3 shl B) ELSE (AB<-3 shl AB),
                      IF (SH=1) THEN (C3<-countup C3),
                      C1<-countdn C1, C2<-countdn C2,
/D'*RUN*P(3)*F(24)/   IF (C2=0) THEN (CAR<-countup CAR,
                         IF (MC(0)+MC(1)+MC(2)=1) THEN (D<-1)),
                      IF ((C1=0)*(C2≠0)*(C3=12)) THEN (CAR<-F(ADS),
                         IF (MC(0)+MC(1)+MC(2)=1) THEN (D<-1)), $F(ADS)=65
                      IF ((C1=0)*(C2≠0)*C3≠12)) THEN (CAR<-RETURN)

Comment, load a main-memory-word micro-instruction located at C.M. address 64
```

```
/D'*RUN*P(0)/              F<-CM(CAR),

/RUN*MC(0)*P(1)*DAR(1)/    AR<-X2,

/RUN*MC(0)*P(1)*F(31)/     READ<-1

/D'*RUN*P(1)*DC(4)/        C2<-12,

/D'*RUN*P(2)*DX(2)/        X2<-countup X2,

/D'*RUN*P(3)*F(33)/        IF (MC(0)+MC(1)+MC(2)=1) THEN (D<-1),

Comment, this micro-instruction remains in F until the end of the main
         memory cycle.

/RUN*MC(1)*P(1)/           SR<-M(AR),

/RUN*MC(2)*P(1)DBS(2)/     B(IN)<-SR,

/RUN*MC(2)*P(3)*DT(7)/     CAR<-countup CAR,

/RUN*MC(3)*P(3)*DT(7)/     IF (C1≠0) THEN (CAR<-F(ADS)),        $F(ADS)=63

                           IF ((C1=0)*(C3≠12)) THEN (CAR<-RETURN),

/RUN*MC(3)*P(3)/           D<-0,

Comment, store a main-memory word micro-instruction located at C.M.
         address 65

/D'*RUN*P(0)/              F<-CM(CAR),

/RUN*MC(0)*P(1)*DAR(2)/    AR<-X3,

/RUN*MC(0)*P(1)*F(34)/     WRITE<-1,

/D'*RUN*P(1)*DC(5)/        C3<-0,

/D'*RUN*P(2)*DX(3)/        X3<-countup X3,

/D'*RUN*P(2)*DBS(1)/       SR<-A,

/D'*RUN*P(3)*DT(4)/        CAR<-RETURN,

/RUN*MC(3)*P(1)/           M(AR)<-SR,
```

Register UNPACK in Figure 32 is replaced by register RETURN in the above description. When the unpacking subsequence is called, the next control memory address is stored in register RETURN, and the transfer from the calling sequence to the unpacking subsequence is carried out by micro-operation CAR←F(ADS). Subregister F(ADS) contains the address of the unpacking subsequence in the control memory. When the unpacking subsequence is terminated, the return to the calling sequence is performed by micro-operation CAR←RETURN.

As is shown above, the first micro-instruction is located at control memory address 63. As shown in the sequence chart of Figure 32, in addition to shifting register B of casregister AB, this micro-instruction performs a four-way branch as below,

(a) If condition $(C1\neq0)*(C2\neq0)$ is true, then repeat the shift-and-test micro-instruction;

(b) If condition $(C2=0)$ is true, then a word in the input buffer is read out of the main memory and stored in subregister B(IN);

(c) If condition $(C1=0)*(C2\neq0)*(C3=12)$ is true, then the contents in register A are written into the output buffer in the main memory;

(d) If condition $(C1=0)*(C2\neq0)*(C3\neq12)$ is true, then the unpacking subsequence is terminated, and the control is returned to the calling sequence.

If the branch (b) or (c) is performed, the wait register D is set to one because the micro- instructions to be executed require one main memory cycle.

The second micro-instruction is located at control memory address 64. It is fetched and executed if condition $(C2=0)$ of the shift and test micro-instruction is true. This micro-instruction reads a word from the input buffer in the main memory and stores it into subregister B(IN). The micro-instruction requires one main memory cycle. To accomplish this, wait register D is set to one, causing the micro-instruction to remain in register F for a full main memory cycle. Upon completion of the micro-instruction, a three-way branch is performed:

(a) If the condition $(C1 \neq 0)$ is true, the shift and test micro-instruction is performed;

(b) If the condition $(C1=0)*(C3 \neq 12)$ is true, then the unpacking subsequence is terminated, and the control is returned to the calling sequence.

(c) If neither of the above is true, then the condition $(C1=0)*(C2=12)$ must be true. In this case, the contents of register A are written into the output buffer in the main memory.

The third micro-instruction is located at control memory address 65. It is executed when the condition $(C1=0)*(C3=12)$ is found to be true in either of the first two micro-instructions. The micro-instruction causes the contents of register A to be written to the output buffer in the main memory with only one control memory cycle which occurs at the beginning of the main memory cycle. During this control cycle, the write to the output buffer is initiated, and it is completed at three control memory cycles later. Again, the micro-instruction returns control to the calling sequence.

5.8.2 Initialization sequence

The initialization sequence initializes the sequences. The three micro-instructions that make up the sequence are shown below.

Comment, Initialization sequence (12)

Comment, initiate read of input-buffer micro-instr. located at C.M. address 66

/D'*RUN*P(0)/ F<-CM(CAR)

/RUN*MC(0)*P(1)*DAR(1)/ AR<-X2

/RUN*MC(0)*P(1)*F(31)/ READ<-1

/D'*RUN*P(1)*DBS(5)/ B(ADR)<-X7

/D'*RUN*P(1)*DC(6)/ C3<-7

/D'*RUN*P(1)*F(27)/ A<-0

/D'*RUN*P(3)*DT(1)/ CAR<-countup CAR

Comment, main-memory-read micro-instruction located at C.M. address 67

/D'*RUN*P(0)/ F<-CM(CAR)

/RUN*MC(1)*P(0)/ SR<-M(AR)

/D'*RUN*P(1)*DC(4)/ C2<-12

/D'*RUN*P(2)*DX(2)/ X2<-countup X2

/D'*RUN*P(3)*DT(1)/ CAR<-countup

Comment, load register B micro-instruction located at C.M. address 68

/D'*RUN*P(0)/ F<-CM(CAR)

/RUN*MC(2)*P(1)*DBS(2)/ B(IN)<-SR

/D'*RUN*P(1)*DC(3)/ C1<-5

/D'*RUN*P(3)*DT(1)/ CAR<-countup CAR

The three micro-instructions located at control memory addresses 66, 67, and 68, execute sequentially during the first three control memory cycles of a main memory cycle. They initialize the contents of registers A and B and set the counters for the fetch sequence. The third micro-instruction increments control memory address register CAR to begin the fetch sequence.

5.8.3 Fetch sequence

The fetch sequence performs the fetch of the next relocatable word. The two micro-instructions of the fetch sequence appear below.

```
Comment, Fetch sequence                                                    (13)
Comment, initiate-unpacking subsequence micro-instr. located at C.M. address 69
/D'*RUN*P(0)/              F<-CM(CAR)
/D'*RUN*P(2)*F(35)/        SH<-1
/D'*RUN*P(2)*F(26)/        RETURN<-CAR
/D'*RUN*P(3)*F(26)/        RETURN<-countup RETURN
/D'*RUN*P(3)*DT(2)/        CAR<-F(ADS)              $F(ADS)=63
Comment, decode and initiate unpacking subseq. micro-instr. at C.M. address 70
/D'*RUN*P(0)/              F<-CM(CAR)
/D'*RUN*P(1)*F(25)/        OP<-B(1-3), FS<-B(4-7), II<-B(8-9)
/D'*RUN*P(1)*DC(2)/        C1<-3
/D'*RUN*P(2)*F(32)/        SH<-0
/D'*RUN*P(2)*F(26)/        RETURN<-CAR
/D'*RUN*P(3)*F(26)/        RETURN<-countup RETURN
/D'*RUN*P(3)*DT(2)/        CAR<-F(ADS)              $F(ADS)=63
```

As shown above, the first micro-instruction is located at control memory address 69. The function of the micro-instruction is to perform the transfer to the unpacking sequence with the indication to perform a shift of casregister AB. This is accomplished by loading register RETURN with the address of the next micro-instruction, setting register SH to 1, and loading the control address register with the address of the unpacking subsequence.

The second micro-instruction is executed upon return from the unpacking sequence. This micro-instruction loads the registers OP, FS, and

II. It then calls the unpacking sequence with register SH set to zero in order to left-adjust the address or data part of the relocatable word in register B. It loads the register RETURN with the address of the next micro-instruction, which is the first micro-instruction of the address modification sequence.

5.8.4 Address modification sequence

The address modification sequence is described in the form of a sequence chart in Figure 33. The CDL description of the sequence appears below.

```
Comment, Address modification sequence                                  (14)
Comment, branch on OP micro-instruction, located at C.M. address 71
/D'*RUN*P(0)/            F<-CM(CAR)
/D'*RUN*P(1)*DBS(6)/     X8<-B(ADR)
/D'*RUN*P(3)*DT(3)/      CAR<-F(1-5)-OP      $F(ADS)=72
/D'*RUN*P(3)*F(33)/      IF(MC(0)+MC(1)+MC(2)=1) THEN(D<-1)
Comment, error-stop micro-instruction, located at C.M. address 72
/D'*RUN*P(0)/            F<-CM(CAR)
/D'*RUN*P(3)*F(36)/      RUN<-0
Comment, data micro-instruction, located at C.M. address 73
/D'*RUN*P(0)/            F<-CM(CAR)
/D'*RUN*P(1)*DC(1)/      C1<-FS
/D'*RUN*P(3)*DT(2)/      CAR<-F(ADS)         $F(ADS)=69
Comment, relative address micro-instruction, located at C.M. address 74
/D'*RUN*P(0)/            F<-CM(CAR)
/D'*RUN*P(1)*DBS(8)/     B(ADR)<-X8 add X7
/D'*RUN*P(1)*DC(1)/      C1<-FS
/D'*RUN*P(3)*DT(2)/      CAR<-F(ADS)         $F(ADS)=69
```

Comment, common area address micro-instruction, located at C.M. address 75

```
/D'*RUN*P(0)/              F<-CM(CAR)
/RUN*MC(0)*P(1)*DAR(3)/    AR<-X8 add X4
/RUN*MC(0)*P(1)*F(31)/     READ<-1
/D'*RUN*P(3)*DT(2)/        CAR<-F(ADS)        $F(ADS)=81
```

Comment, external address micro-instruction, located at C.M. address 76

```
/D'*RUN*P(0)/              F<-CM(CAR)
/RUN*MC(0)*P(1)*DAR(4)/    AR<-X8 add X5
/RUN*MC(0)*P(1)*F(31)/     READ<-1
/D'*RUN*P(3)*DT(2)/        CAR<-F(ADS)        $F(ADS)=80
/D'*RUN*P(3)*F(33)/        IF(MC(0)+MC(1)+MC(2)=1) THEN (D<-1)
/RUN*MC(1)*P(1)/           SR<-M(AR)
/RUN*MC(2)*P(1)*DBS(3)/    X8<-SR(AD)
/RUN*MC(3)*P(3)/           D<-0
```

Comment, stop micro-instruction, located at C.M. address 77

```
/D'*RUN*P(0)/              F<-CM(CAR)
/D'*RUN*P(3)*F(36)/        RUN<-0
```

Comment, error-stop micro-instruction, located at C.M. address 78

```
/D'*RUN*P(0)/              F<-CM(CAR)
/D'*RUN*P(3)*F(36)/        RUN<-0
```

Comment, error-stop micro-instruction, located at C.M. address 79

```
/D'*RUN*P(0)/              F<-CM(CAR)
/D'*RUN*P(3)*F(36)/        RUN<-0
```

Comment, read from main-memory-table micro-instruction, located at C.M. address 80

```
/D'*RUN*P(0)/              F<-CM(CAR)
/RUN*MC(0)*P(1)*DAR(5)/    AR<-X8 add X6
/RUN*MC(0)*P(1)*F(31)/     READ<-1
/D'*RUN*P(3)*DT(1)/        CAR<-countup CAR
```

Comment, branch if no increment micro-instruction, located at C.M. address 81

/D'*RUN*P(0)/	F←CM(CAR)
/RUN*MC(1)*P(1)/	SR←M(AR)
/D'*RUN*P(3)*DT(5)/	IF(II=0) THEN (CAR←F(ADS)) ELSE (CAR←countup CAR) $F(ADS)=86

Comment, initiate unpacking subseq. micro-instr., located at C.M. address 82

/D'*RUN*P(0)/	F←CM(CAR)
/RUN*MC(2)*P(1)*DBS(3)/	X8←SR(AD)
/D'*RUN*P(1)*DC(1)/	C1←FS
/D'*RUN*P(2)*F(32)/	SH←0
/D'*RUN*P(2)*F(26)/	RETURN←CAR
/D'*RUN*P(3)*F(26)/	RETURN←countup RETURN
/D'*RUN*P(3)*DT(2)/	CAR←F(ADS) $F(ADS)=63

Comment, increment-decrement branch micro-instr., located at C.M. address 83

/D'*RUN*P(0)/	F←CM(CAR)
/D'*RUN*P(1)*DBS(4)/	X1←B(ADR)
/D'*RUN*P(3)*DT(6)/	IF (II=1) THEN (CAR←F(ADS)) ELSE (CAR←countup CAR) $F(ADS)=85

Comment, 2's complement micro-instruction, located at C.M. address 84

/D'*RUN*P(0)/	F←CM(CAR)
/D'*RUN*P(1)*F(29)/	X1←X1'
/D'*RUN*P(2)*DX(1)/	X1←Countup X1
/D'*RUN*P(3)*DT(1)/	CAR←countup CAR

Comment, modify absolute address micro-instr., located at C.M. address 85

/D'*RUN*P(0)/	F←CM(CAR)
/D'*RUN*P(1)*DBS(7)/	B(ADR)←X8 add X1
/D'*RUN*P(1)*DC(1)/	C1←FS
/D'*RUN*P(3)*DT(2)/	CAR←F(ADS) $F(ADS)=69

Comment, no increment micro-instruction, located at C.M. address 86

```
/D'*RUN*P(0)/              F<-CM(CAR)
/RUN*MC(2)*P(1)*DBS(3)/    X8<-SR(AD)
/D'*RUN*P(1)*F(28)/        X1<-0
/D'*RUN*P(3)*DT(2)/        CAR<-F(ADS)      $F(ADS)=85
```

The first micro-instruction shown above is located at control memory address 71. This micro-instruction performs an eight-way branch on the contents of register OP. This is accomplished by concatenating the first five bits of subregister F(ADS) with register OP to form an eight-bit control memory address. Of the eight possible addresses, five are legitimate and will be discussed in detail subsequently. The other three, corresponding to OP values 0, 6, and 7, cause an error-stop. The subregister F(ADS) contains the control memory address 72. This means that the eight addresses possible are 72 through 79, with addresses 72, 78, and 79 corresponding to the illegitimate addresses. It should be noted that register X8 is loaded by this micro-instruction.

If OP is equal to one, the micro-instruction at control memory address 73 is executed. As this means that the relocatable word contains data, the micro-instruction simply loads counter C1 and branches to the fetch sequence.

If OP is equal to two, indicating a relative address, the micro-instruction at control memory address 74 is executed. This micro-instruction adds the address in X8 to the subprogram address in X7 and branches to the fetch sequence.

If OP is equal to three, indicating a common area address, the micro-instruction at control memory address 75 is executed. This micro-instruction initiates the read of the Common Symbol Table in the main memory and branches to the micro-instruction at location 81 to test for the existence of an increment field in the relocatable word. It should be noted that the first micro-instruction sets wait register D in order to assure that this instruction is performed at the beginning of a main memory cycle.

If OP is equal to four, indicating an external address, the micro-instruction at control memory address 76 is executed. This micro-instruction performs a read of the Undefined Symbol Table in the main memory and then branchs to the micro-instruction at control memory address 80 which initiates the read of the Defined Symbol Table in the main memory.

If OP is equal to five, indicating termination of the translation process, the micro-instruction at control memory address 77 is executed. This micro-instruction sets register RUN to zero to stop generation of all signals.

The remaining micro-instructions in control memory address locations 81 through 86 are executed if OP is equal to 3 or 4. These micro-instructions perform the test for the increment or decrement and the address modification operations described in the sequence chart in Figure 33. This entails initializing the unpacking sequence in the micro-instruction at control memory address 82. It should also be noted that the main memory read initiated in the micro-instruction located at the control memory address 75 (OP=3) or 80 (OP=4) is continued in parallel with the execution of these micro-instructions.

5.9 Microprogram

The microprogram is shown in Table 4 where the micro-instructions are stored in the arbitrarily chosen locations 63 to 86. All numbers are binary. Each micro-instruction of the microprogram is obtained from the previous CDL descriptions of the micro-instructions.

For example, the micro-instruction for loading a main memory word described previously in the unpacking subsequence is assigned to location 64. The field F(ADS) should contain 63 because of the branch micro-operation CAR$\leftarrow$F(ADS) in the micro-instruction. In each of the execution statements, there is one micro-operation and a corresponding decoder terminal or control bit in the label according to the control word format in Table 3. Consider the second execution statement of this example. This statement contains micro-operation AR$\leftarrow$X2. According to the control word format, this micro-operation is controlled by decoder terminal DAR(1) in field F(14-16); therefore, this field ,as shown in location 64 of Table 4, contains 001. Similarly, each non-zero subfield controls the execution of one micro-operation or a group of micro-operations by the terminal. Consider also the micro-operation for initiating a memory read READ$\leftarrow$1. According to the control word format, this micro-operation is controlled by control bit F(31). The control word at location 64 in Table 4 shows that this bit is 1. Similarly, each control bit in field F(24-36) controls the execution of one micro-operation or a group of micro-operations. In this manner, the microprogram in Table 4 is prepared.

At location 71 of this microprogram, the micro-instruction branches to one of locations 72 through 79. Locations 72 through 79 contain eight micro-instructions (only five are used), each of which performs the micro-operations required by the OP code of the relocatable word. This table is located by the address in field F(ADS) (see 01001000 at location 71 in Table 4).

It is possible to estimate the speed of translating the executable code (i.e., instructions or data words). At the one extreme, it requires

Table 4 Microprogram for the translation of relocatable code to executable code

control memory address	F(ADS)=F(1-8)	DC(0-7)	DX(0-3)	DAR(0-7)	DBS(0-15)	DT(0-7)	F(24-36)
63	0 1 0 0 0 0 0 1	0 0 0	0 0	0 0 0	0 0 0 0	0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0
64	0 0 1 1 1 1 1 1	1 0 0	1 0	0 0 1	0 0 1 0	1 1 1	0 0 0 0 0 0 0 1 0 1 0 0 0
65	0 0 0 0 0 0 0 0	1 0 1	1 1	0 1 0	0 0 0 1	1 0 0	0 0 0 0 0 0 0 0 0 0 1 0 0
66	0 0 0 0 0 0 0 0	1 1 0	0 0	0 0 1	0 1 0 1	0 0 1	0 0 0 1 0 0 0 1 0 0 0 0 0
67	0 0 0 0 0 0 0 0	1 0 0	1 0	0 0 0	0 0 0 0	0 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0
68	0 0 0 0 0 0 0 0	0 1 1	0 0	0 0 0	0 0 1 0	0 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0
69	0 0 1 1 1 1 1 1	0 0 0	0 0	0 0 0	0 0 0 0	0 1 0	0 0 1 0 0 0 0 0 0 0 0 1 0
70	0 0 1 1 1 1 1 1	0 1 0	0 0	0 0 0	0 0 0 0	0 1 0	0 1 1 0 0 0 0 0 1 0 0 0 0
71	0 1 0 0 1 0 0 0	0 0 0	0 0	0 0 0	0 1 1 0	0 1 1	0 0 0 0 0 0 0 0 0 1 0 0 0
72	0 0 0 0 0 0 0 0	0 0 0	0 0	0 0 0	0 0 0 0	0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1
73	0 1 0 0 0 1 0 1	0 0 1	0 0	0 0 0	0 0 0 0	0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0
74	0 1 0 0 0 1 0 1	0 0 1	0 0	0 0 0	1 0 0 0	0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0
75	0 1 0 1 0 0 0 1	0 0 0	0 0	0 1 1	0 0 0 0	0 1 0	0 0 0 0 0 0 0 1 0 0 0 0 0
76	0 1 0 1 0 0 0 0	0 0 0	0 0	1 0 0	0 0 1 1	0 1 0	0 0 0 0 0 0 0 1 0 1 0 0 0
77	0 0 0 0 0 0 0 0	0 0 0	0 0	0 0 0	0 0 0 0	0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1
78	0 0 0 0 0 0 0 0	0 0 0	0 0	0 0 0	0 0 0 0	0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1
79	0 0 0 0 0 0 0 0	0 0 0	0 0	0 0 0	0 0 0 0	0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1
80	0 0 0 0 0 0 0 0	0 0 0	0 0	1 0 1	0 0 0 0	0 0 1	0 0 0 0 0 0 0 1 0 0 0 0 0
81	0 1 0 1 0 1 1 0	0 0 0	0 0	0 0 0	0 0 0 0	1 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0
82	0 0 1 1 1 1 1 1	0 0 1	0 0	0 0 0	0 0 1 1	0 1 0	0 0 1 0 0 0 0 0 1 0 0 0 0
83	0 1 0 1 0 1 0 1	0 0 0	0 0	0 0 0	0 1 0 0	1 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0
84	0 0 0 0 0 0 0 0	0 0 0	0 1	0 0 0	0 0 0 0	0 0 1	0 0 0 0 0 1 0 0 0 0 0 0 0
85	0 1 0 0 0 1 0 1	0 0 1	0 0	0 0 0	0 1 1 1	0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0
86	0 1 0 1 0 1 0 1	0 0 0	0 0	0 0 0	0 0 1 1	0 1 0	0 0 0 0 1 0 0 0 0 0 0 0 0

7 main memory cycles to translate a data word. At the other extreme, it requires 24 main memory cycles to translate a two-address instruction where the addresses are external addresses. (It requires four relocatable words for one such instruction.) Let the main memory cycle time be 1 microsecond. Then, the microprogram controlled translator is capable of producing from 41,700 to 143,000 instructions or data words per second.

6. References

(1) K.E. Iverson, "A Programming Language", John Wiley and Sons, N.Y., 1962.

(2) A. D. Falkoff and K.E. Iverson, "A Formal Description of System/360", IBM Systems Journal, Vol. 3, No. 3, 1964, pp. 198-263.

(3) H.P. Schlaeppi, "A Formal Language for Describing Machine Logic, Timing and Sequencing (LOTIS)", IEEE Trans. on Electronic Computers, August 1964, pp.439-448.

(4) A. P. Mullery, "A Procedure Oriented Machine Language", IEEE Trans. on Electronic Computers, August 1964, pp. 449-455.

(5) R. M. Proctor, "A Logic Design Translator Experiment Demonstrating Relationships of Language to Systems and Logic Design", IEEE Trans. on Electronic Computers, August 1964, pp. 422-430.

(6) H. Schorr, "Computer-aided Digital System Design and Analysis Using a Register Transfer Language", IEEE Trans. on Electronic Computers, Dec. 1964, pp. 730-737.

(7) Y. Chu, "An Algol-like Computer Design Language", Comm. of the ACM, Oct. 1965, pp. 607-615.

(8) D. L. Parnas, "A Language for Describing the Function of Synchronous Systems", Comm. of ACM, Feb. 1966, pp. 72-76.

(9) J. A. Wilber, "A Language for Describing Digital Computer", Report No. 197, Dept. of Comp. Science, U. of Illinois, Feb. 15, 1966.

(10) A. Giese, "Hargol - A Hardware Oriented Algol Language", Internal Report No. VA5, August 1966, A/S Regnecentralen, Copenhagen, Denmark.

(11) G. Metze and S. Seshu, "A Proposal for a Computer Compiler", Proc. of the SFCC Conference, 1966, pp. 253-263.

(12) M.S. Zucker, "LOCS: An EDP Machines Logic and Control Simulator", International ConventionRecord, Part 3, 1965, pp. 28-50.

(13) M. A. Breuer, "General Survey of Design Automation of Digital Computer", Proc, of the IEEE, Vol. 54, No. 12, Dec. 1966, pp.1708-1721.

(14) Y.Chu and A. Frank, "Symbolic Design of Bi-Tran Six Computer", Tech. Report, TR-66-36, Computer Science Center, U. of Maryland, Nov. 1966.

(15) G. Sardarian, "Symbolic Design for the CDC 1700 Computer Logic", Thesis, Dept. of Electrical Eng., U. of Maryland, Jan. 1967.

(16) C. K. Mesztenyi, "Computer Design Language, Simulation and Boolean Translation." Tech. Report 68-72, Computer Science Center, U. of Maryland, June, 1968.